PROGRAMMING AISSIGNMENT, ECE 695-009

Raunaq Nandy Majumdar

My best performing implementation in GPU for batch size **(N=128)** takes **0.41s**(**414.2 ms)** (kernel + memory copy time) or **0.27s**(**270.8 ms)(**kernel time) to complete all the 5 convolution layer of alex-net implementation kernel whereas CPU takes **1035s** for N=128 to complete the same thus GPU gives a speedup of **3696.5x** times speedup over CPU for just kernel execution and around **2490x** speedup when data migration overhead is also considered. For N=32, max speedup is approx. 2447x and for N=1 max speedup is 2660x.

Steps and Optimization and Analysis:

1. I first created the CPU benchmarks for all the 5 layers for N=1,32,128 and stored them in file for comparison purpose with GPU results
2. Then implemented the unified memory GPU implementation followed by cudamemcpy implementation.
3. Further optimizations were done on speeding up the code on top of the cudamemcpy implementation. Optimizations include using shared memory for storing weight matrix, using more registers to reduce access to global memory, launching 2 kernels to exploit more parallelism, reducing branch divergence in a block, exploiting coalesced access by threads, using vector loads, using half precision data to improve memory bandwidth. Optimizations were chosen to be implemented on cudamemcpy version over unified memory version because for larger batch sizes even with memcpy overhead it is faster than unified memory because the later has lot of page faults leading to migration overhead which can be handled with either prefetching the data first or initializing the data first in GPU by launching another kernel.
4. If we talk about the way I parallelized the execution, for M weights, batch size N the total number of output pixels are NxMxExF where each thread block in the grid compute ExF outputs and there are NXM such blocks, N in x-dim and M in y-dim In grid where block dim is EXF, so necessarily each thread compute a single output coordinate. **(fig 2a)(table 1, table 2)**
5. The max thread block size is 1024 for GV100 so as per the technique discussed above **for layer-1 where 55x55 is the output block size, we need to parallelize it differently by launching blocks of 28x28 size and letting each thread compute 2 output pixels**. This can be done in 2 ways, (coalesced and non coalesced way) **(fig9, fig10)**. Letting each thread compute 2 pixels lead to minimum divergence as at the end tid 27 will compute 2 extra pixels which is handled using a small "if" condition which will create a branch divergence with 27 taken and 1 not taken threads. If we assume the scheduler uses majority issue heuristic then 27 threads in the taken path will be given priority leading to faster execution time. **(table 2)**
6. For **N=128**, the unified memory **(UM)** version gives a total speedup of **873x** and memcpy **(Mcpy)** version gives a total speedup of **895x(considering with memcpy time). UM** is slow because it incurs page faults. It can be made faster if we somehow prefetch the data into the GPU memory beforehand.
7. First optimization **(rla2m)** was done to reduce the continuous global LD-ST dependency of the output array by computing intermediate results and storing into thread private registers before finally accessing global output array once just to store the result where threads with consecutive tid access consecutive location thus reducing LD-ST memory dependency in global output array. This gives a total speedup of **1145x (N=128)** when we use coalesced access for layer 1. We can clearly see form **fig 11** and **table 8** that coalesced access improves the performance of layer 1 by **1.7x** over non-coalesced accesses because as **fig 9** suggests consecutive threads accessing consecutive memory location leads to better memory BW utilization and this idea is applicable for all other optimizations.
8. As all threads in a block use same weight matrix to compute the output it can be put into shared memory thus reducing global access. The first implementation **(rswbdla2m)** allows only one thread to load weights from global to shared memory while other threads from that block wait in the barrier thus creating high branch divergence and allowing only single thread load data thus necessarily serializing the accesses to global memory and underutilizing the BW, nevertheless causing speedup over baseline memcpy case by **977x (N=128)** but it is lesser than earlier case as all threads have to wait at barrier without doing anything while only 1 thread per block fetches data from global to shared memory thus decreasing efficiency over last optimization.
9. If instead of letting only 1 thread load data from global to shared memory if we allow RXS threads to load corresponding weight values across all the channels for layer 2,3,4,5 and RXSXC threads for layer 1 to load weight values from global memory then we can reduce thread divergence. It also reduces amount of work done by a single thread and thus improves efficiency of the system. **This different handling of layer 1** is because for them RxSxC < EXF. This optimization **(rswmtla2m)** gives a total speedup of **1171x** over CPU for **N=128** and it is faster than previous two optimizations.
10. The previous two optimizations use both shared memory and registers for computation. This optimization **(swso)** only uses shared memory for weight and also to temporarily store the intermediate MAC value. And it only uses 1 thread to load the weights to shared memory so it can be considered as an experiment on top of **rswbdla2m**. I expected the performance speedup to be less than **rla2m** because registers are faster than shared memory, we are only using 1 thread to load weights to shared memory and as of now only putting weights into shared memory which is really small in number so not much improvement can be expected over **rswbdla2m** also .The total speedup in this case is **1000x** for **N=128**.

11. Till now we only parallelized N and M. But if we analyze clearly then we can also parallelize C because all the channels are computing carefully. Now to handle this I invoked 2 kernels. Block size for both the kernel remains same but grid size of first kernel which does **element wise matrix mul** is now **NxMxC (table 1), (fig 2b)** and after it computes it result I invoke another kernel which does the **element wise reduction of all the products across C channels. Its grid size is NxM**. This actually allows all threads to do less work thus reduces computation time with slight increase in memcpy time **(table 3)**. For this multiple kernel approach **(Mk)** without any shared memory optimization we get a speedup of **1015x** for **N=128** speed up over CPU which is also higher than baseline memcpy implementation.
12. Now adding shared memory for weight with less branch divergence **(Mk-sw-mt)** on top of the multiple kernel feature we get a speedup of **1535x** for **N=128**. This happened due to obvious reasons that shared memory is faster than global memory and now that we have less shared memory per block but same number of threads per block as the single kernel implementation discussed above, now while accessing shared memory chances of bank conflict in shared memory is less in multi-kernel approach. Please note that while we are observing speedup using multiple kernel approach till now for layer 1 accesses to memory are non-coalesced. Coalesced access for multi-kernel approach for layer 1 is implemented in next few optimizations.
13. We can also put entire input of size HxW for all layer except 1 into shared memory now that we have parallelized for N,M,C **(Mk-sw-mt-si-c)**. We cannot put inputs for layer 1 entirely because it overshoots the shared memory limit in the SM. So to speed up layer 1 over optimization in **Mk-sw-mt** we added coalesced memory access in layer 1 instead of non-coalesced one in **Mk-sw-mt** and measured the speed up. These optimizations **(Mk-sw-mt-si-c)** gives a speed up of **1950x** for **N=128**. From **fig 11, table 8** it can be seen that execution time of layer 1 sped up by **36%** which will scale up for even higher batch sizes.
14. Now if we use vector loads like float3, float4 etc to load from memory (both shared and global wherever necessary) we can remove one additional loop thus removing branch instructions thus getting some additional speedup. Also using float3 is faster than using 3 float so we get an edge there also. For layer 3,4,5 weight matrix is 3x3 so if we use float3 for them then we can remove one additional for loops. For layer 2 and 1 where weight matrix are 5x5 and 11x11 respectively we can use float3 to properly unroll the loop thus removing backward branches which gives us additional speedup in those layers. This optimization **(Mk-sw-mt-si-c-vec)** gives a speedup of **2490x** for **N=128** when data copying overhead is also considered along with kernel execution time.
15. I also tried out half precision floating point for both the single kernel **(half-precision)** implementation. Adding half precision floating point for baseline memcpy implementation gave a speedup of **1060x** for **N=128** speedup over CPU and **1.2x** times speedup over baseline memcpy implementation because half-precision floating point leads to better data packing in memory thus utilizes BW efficiently.
16. Adding half precision floating point on multi-kernel and shared input shared weight optimization gives a total speedup of **2031x (Mk-sw-mt-si-c-half)** over CPU for **N=128**. Half precision floating point creates 16 bit floating points thus leading to denser data packing in memory thus leading to better utilization of memory BW.
17. The issue with multiple kernel approach is **allocates a lot of memory in global memory in GPU** (C*N*M*E*F*4 bytes of output array by kernel 1) so it can fail to allocate memory for larger batch sizes for devices having smaller global memory. It can also fail if multiple to do cudaMalloc when multiple memory intensive processes are running on GPU**. So while running the multiple kernel implementation one has to make sure that sufficient global memory is available and no other memory intensive process is running at that time**. One way to handle this is for higher batch sizes (N>128) launch using one kernel only, do a local reduction in that kernel only. A small experiment on the same has been done on all the layers and its results with comparison with optimization **Mk-sw-mt-si-c-vec** is plotted.**(fig 15)** It will achieve a little speedup in layer 3,4,5 (around **2ms**)for larger batches but for smaller batches there is high precision loss and execution time is also more. For smaller batches execution time is more because if we are doing local reduction across different blocks it has to be done in global memory due to the large size of the output array, so we have to do lot of global memory load and store hence having larger batches amortizes that latency but smaller batches are unable to hide that. Another way of handling this memory bottleneck is we can partially parallelize C in kernel 1, for example if C=256, we can parallelize 128 of them and let each thread-block compute outputs for 2 channels in kernel 1 and let kernel 2 do the reduction job as usual. You can also use unified memory features for multiple kernel as it does not explicitly copy data, it only migrates pages when needed but it takes a lot of time for larger batches. So here we have a tradeoff between speedup and global memory limit.

Set up and Methodology:

1. CPU timings were measured once at the beginning only. I have used my CPU time for N=1,32,128**(Table 4,5).**
2. **All codes were executed when there were no other process running on the GPU. This step is critical for measuring timings of multi-kernel implementation as it is bound by global memory allocation.**
3. All the code has a serial implementation of the convolution layer in it too. The function call is commented from the main function
4. In main function GPU output for each index is compared with corresponding CPU output and the maximum error is updated accordingly which the code prints at the end
5. To save space in my scholar account I am not printing or storing any GPU computed values of the convolution as the files take a lot of space for larger batches. The max error (epsilon) provides a good estimation whether the computation is right or wrong. Mostly the epsilon is in range of **0.0xxx to 0.000xxx**.

6. Having said that there are print statements inside the last loop in main function which lets you print GPU results for each index. It is commented out as of now.

Additional Optimization Analysis:

1. Speedup is most for all the optimizations for N=1 as number of blocks launched here is less so wait time for other blocks is also less. For larger batches there are many waiting thread blocks due to limited resources hence even though the optimizations gives appreciable speedup it is less than N=1.**(Fig 3a,4a,5a,6a,7a)**
2. It can be seen from **fig 3a,4a,5a,6a,7a**, for larger batches **Mk-sw-mt-si-c-vec** gives the best speedup. . It stores weights and inputs into shared memory for each block for layers 2,3,4,5 and only shared weights and coalesced accesses for layer 1 with vectored loads. Only for layer 1, **Mk-sw-mt-si-c-half** gives best speedup because better BW-efficiency due to half precision wins as layer 1 has less parallelism than other layers. And as we cannot put input matrix into shared memory for layer 1, better BW efficiency and data packing in half precision optimization earns an edge in performance.
3. As for layers 2,3,4,5 when our block size is ExF and as consecutive threads execute consecutive memory locations, by default all the accesses to memory are coalesced for these layers. For layer 1 as we have launched 28x28 thread block for 55x55 output block each thread computes 2 output pixels with the exception of last thread hence this minimum branch divergence is not detrimental. Computing 2 pixels by each thread can be done using either coalesced or non-coalesced accesses. We implemented both coalesced and non-coalesced layer 1 for certain optimizations and plotted their performance comparison in **Fig 11**.
4. If we analyze **fig 12a, 12b,12c** we can see especially for layer 1 (N=128) data copying time is more than kernel execution time obviously due to higher dimension of output matrix. We have analyzed for 3 of our best case optimizations in order of degrading performance, and the same analysis can be done for other optimizations also. I think the data copying overhead can be reduced if we use pinned memory for memory allocation wherever necessary.
5. If we observe graphs in **Fig 3b,4b,5b,6b,7b,** we can see as kernel execution time reduces with better and better optimization, memory copy time still remains same thus for even better optimizations in kernel execution, data migration overhead can become a bottleneck unless handled carefully.
6. One small observation is if we compare optimizations **swso** and **rswbdla2m** for **N=128**, we see that the former one outperforms the later for layer 4 and 5 and underperforms for layer 1,2,3 , which I suppose is as number of threads in layer 1,2,3 are more than 4 and 5 that's why the fact that registers are faster than shared memory has higher effect for layer 1,2,3 and as layer 4 and 5 has smaller thread blocks this effect is not so prominent.
7. Performance of all layers for N=1, 32,128 are plotted for **rswmtla2m** and **Mk-sw-mt-si-c-vec** optimization. As discussed later layer 2 achieves most speedup due as its CPU implementation is very slow so massive parallelism helps here. **(Fig 13a,13b)**
8. I did an experiment for kernel execution time for one of the single kernel optimization **(rswmtla2m)** and observed that kernel execution time in GPU scales somewhat linearly with batch size **(Fig 14a)**. **Fig 14b** shows effect of scaling batch sizes to even higher values on speedup. If we observe the kernel execution time and speedup then we can understand that execution time increases almost linearly with batch size byt memory copying overhead becomes bottleneck in the speedup.
9. One bottleneck for scaling the multi-kernel optimization for best case situation **(Mk-sw-mt-si-c-vec)** or rather any multi-kernel implementation is it allocates a lot of global memory so for smaller devices it becomes difficult to allocate memory while launching kernel. So the scaling experiment is done with single kernel, less precise experimental **(exp-Mk-sw-mt-si-c-vec**) implementation. Its preciseness is good for higher batches but for smaller batches both speedup and preciseness is less. For layers 1,2 speedup using the lesser precise experimental version is less than more precise version. **Fig 16** and **table 11** tabulates our experiment for speedup measurement for less precise and more precise version. **Fig 15a** shows execution time scaling and **Fig 15b** shows speedup scaling with larger batch size for the less precise experimental version. As analyzed above for larger batches, memory-copying time becomes a bottleneck for speedup.
10. Some common observations from the optimizations are that using more registers instead of global memory access increases performance.
11. Coalesced accesses are faster as they utilize memory BW efficiently.
12. It has been observed that layer 2 has benefitted the most from the speedup. It can be attributed to the product of MxExF. It is maximum for layer 2 that's why it achieves maximum speed up for all batches. Also for the same reason CPU execution time for layer 2, N=128 was highest.
13. Only using 1 thread to load weights from global to shared memory is a bad optimization as other threads are waiting in the barrier because if batch size is increased then we might not see high performance
14. Half precision floating point operation was little tough to implement in vector load optimization which also happens to be my best case optimization. Because while loading data using vector loads we have to make sure that the data are properly aligned to avoid loading wrong data. Incorporating half precision on top of that means using float3 pointer on half data type will cause us to fetch unwanted data thus giving us wrong results
15. Precision of GPU with respect to CPU result is better when we use 32-bit floating point. We lose precision when we use half precision floating points for GPU computation.

Appendix

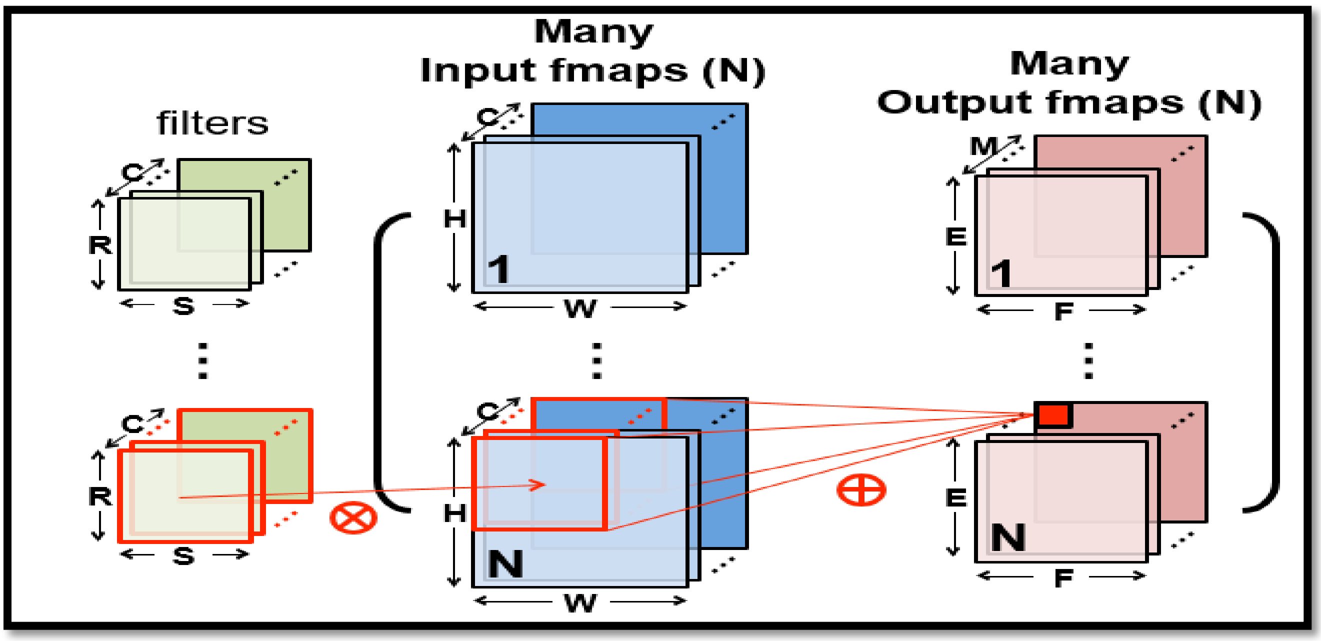
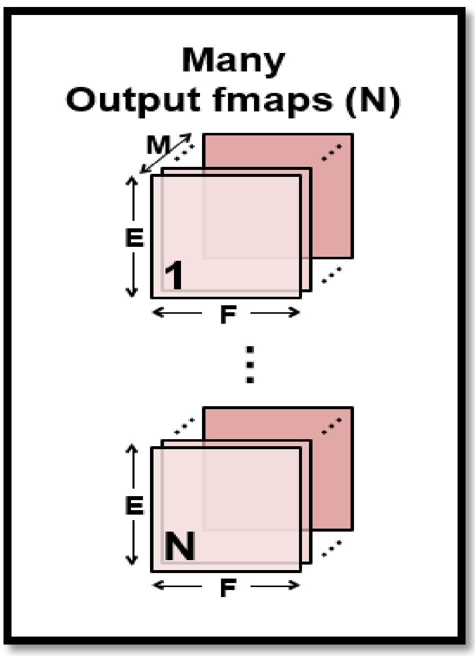


Fig1: Convolution layer operation where output size is NxMxExF



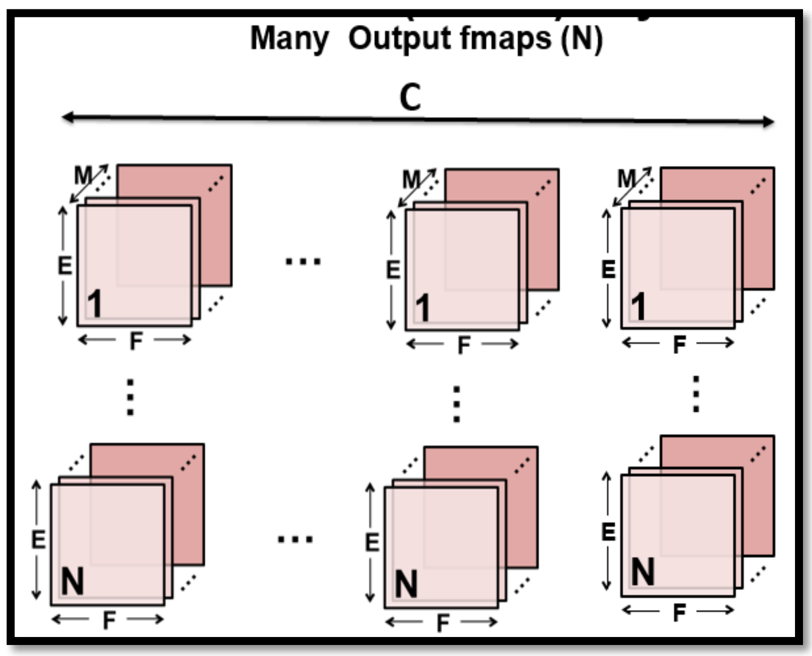


Fig 2a: Single kernel parallelization (NxM) Fig2b: Multi-kernel parallelization (grid size: NxMxC)

Abbreviation of the optimization	Grid dim	Explanation of the optimization
Mcpy	NxM	Cudamemcpy, lots of access to global memory
UM	NxM	Mallocmanaged using unified memory, lots of access to global memory
rla2m	NxM	Cudamemcpy, more registers, less access to global memory
rswbdla2m	NxM	Cudamemcpy, more registers, shared weight ,more branch divergence, less access to global memory
rswmtla2m	NxM	Cudamemcpy, more registers, shared weight, less branch divergence, less access to global memory
Swso	NxM	Cudamemcpy, moderate #registers, shared weight ,more branch divergence,shared output less access to global memory
Halfprecision	NxM	Cudamemcpy, half precision, lots of access to global memory
Mk	NxMxC	Cudamemcpy, multiple kernel lots of access to global memory, more memory
Mk-sw-mt	NxMxC	Cudamemcpy, multiple kernel more registers, shared weight, less branch divergence, less access to global memory, more memory
Mk-sw-mt-si-c	NxMxC	Cudamemcpy, multiple kernel more registers, shared weight, shared input less branch divergence, less access to global memory, coalesced access for layer 1, more memory
Mk-sw-mt-si-c-vec	NxMxC	Cudamemcpy, multiple kernel more registers, shared weight, shared input less branch divergence, less access to global memory, coalesced access, vector load, more memory
Mk-sw-mt-si-c-half	NxMxC	Cudamemcpy, multiple kernel more registers, shared weight, shared input less branch divergence, less access to global memory, coalesced access, half precision
exp-Mk-sw-mt-si-c-vec, single kernel	NxMxC	Cudamemcpy, single kernel more registers, shared weight, shared input less branch divergence, less access to global memory, coalesced access, vector load but less precise but uses less memory

Table 1: Abbreviations describing optimizations used for different analysis for performace

Layer	Output array size	Thread block size	Remarks (if any)
1	55x55	28x28	Maximum size of block is 1024 which is less than 55x55 so 28x28 will allow 1 thread to compute 2 output pixels with minimum thread divergence (assuming majority issue heuristic)
2	27x27	27x27	Each thread will compute 1 output pixel, by default coalesced access to memory
3	13x13	13x13	----------------do----------------------------------
4	13x13	13x13	----------------do----------------------------------
5	13x13	13x13	----------------do----------------------------------

Table 2: Block size used for each layer and explanation for the choice

Layer	N	Data migration overhead for cudamemcpy (ms)	Data migration overhead for cudamemcpy ,half prec(ms)	Data migration overhead for cudamemcpy,mk(ms)	Data migration overhead for cudamemcpy half prec, mk(ms)
1	1	0.144	0.1	0.144	0.1
	32	14	11	14	14
	128	60	48	60	57
2	1	0.25	0.11	0.25	0.12
	32	10	7.7	10	9
	128	39	30	42	36
3	1	0.35	0.13	0.37	0.19
	32	5	3.8	5.7	5.2
	128	24	17	26	18
4	1	0.55	0.23	0.55	0.23
	32	4	3.2	4.1	3.2
	128	17	12	17	14
5	1	0.33	0.14	0.35	0.14
	32	3	2.3	3	2.3
	128	12	10	14	10.3

Table 3: Data copying overhead

Layer	N	CPU (s)	Mcpy (ms)	UM (ms)	rla2m (ms)	rswbdla2m (ms)	rswmtla2m (ms)	swso (ms)	halfprecision (ms)
1	1	0.83	1.86	3.44	0.4	0.6	0.4	0.9	1.68
	32	23.95	37.55	47.02	11.3	13.8	11.3	19	34.03
	128	100	149	183.73	48	53	49	83.63	133
2	1	3.8	3.14	6.223	0.7	1.43	0.8	1.61	2.92
	32	108	85	94.89	31.35	47.6	31.32	46.36	79
	128	432	360	407	299	340	281	339	326
3	1	1.3	1	3.88	0.25	0.7	0.3	0.7	0.9
	32	37.63	34.49	49.43	27	35.14	27.4	35	29.31
	128	147	189	244.03	174.5	204	171	204	120
4	1	2	1.8	5.9	0.37	0.8	0.4	0.8	1.13
	32	53.5	44.7	55.22	20.09	26.81	17.8	24.7	42.53
	128	213	181.2	201.52	145	184.73	139.6	155.6	167
5	1	1.3	1.12	3.48	0.35	0.6	0.33	0.6	1.06
	32	36.2	29.78	39.4	13.32	17.93	11.86	16.7	28.43
	128	143	129	148.19	96	123	92	104	113.2

Table 4: Kernel execution time for various optimization for single kernel implementation

Layer	N	Mcpy	UM	rla2m	rswbdla2m	rswmtla2m	swso	halfprecision
1	1	415	241	1673	1152	1596	830	488
	32	469	509	958	887	958	725	532
	128	480	546	925	885	925	700	552
2	1	1120	612	4000	2261	3800	2111	1225
	32	1136	1136	2700	2038	2700	2000	1255
	128	1082	1061	1320	1193	1337	1133	1213
3	1	962	325	2600	1300	2166	1300	1181
	32	964	767	1176	875	1175	940	1175
	128	693	602	757	647	761	647	1065
4	1	952	333	2500	1538	2105	1538	1503
	32	1098	969	2220	1725	2547	1910	1188
	128	1075	1056	1410	1065	1347	1245	1189
5	1	928	371	2600	1368	1969	1444	1083
	32	1131	918	1920	1810	2445	1905	1179
	128	1014	966	1330	1059	1375	1232	1172

Table 5: Performance speedup for various optimizations for single kernel implementation

Laye r	N	CPU (s)	Mk (ms)	Mk-sw-mt(ms)	Mk-sw-mt-si-c (ms)	Mk-sw-mt-si-c-vec(ms)	Mk-sw-mt-si-c-half (ms)
1	1	0.83	1.17	0.45	0.29	*0.27	0.24
	32	23.95	34.15	13.2	8.3	*7.8	7
	128	100	136.66	58	37	*30.6	29.2
2	1	3.8	2.2	0.93	0.9	0.56	0.97
	32	108	76	33	30	15	30
	128	432	302	229	135	69	111
3	1	1.3	0.89	0.48	0.45	0.37	0.5
	32	37.63	28.8	15	14	11	16
	128	147	128	65	63	50	63
4	1	2	1.39	0.7	0.67	0.54	0.67
	32	53.5	42	22	21	16	23
	128	213	178	97	93	71	90
5	1	1.3	0.97	0.48	0.45	0.38	0.45
	32	36.2	29.5	15	14	10	15
	128	143	125	73	64	48	64

Table 6: Kernel execution time for various optimization for multi-kernel implementation

Layer	N	Mk	Mk-sw-mt	Mk-sw-mt-si-c	Mk-sw-mt-si-c-vec	Mk-sw-mt-si-c-half
1	1	643	1456	2024	*2128	2305
	32	499	887	1088	*1141	1140
	128	510	847	1030	*1111	1123
2	1	1583	3454	3454	4750	3454
	32	1270	2571	2769	4500	2769
	128	1263	1600	2511	4235	2823
3	1	1181	1368	1369	1756	1494
	32	1113	1791	1791	2213	1734
	128	973	1670	1709	2013	1709
4	1	1111	1666	1666	1818	1818
	32	1163	2057	2131	2675	1981
	128	1097	1884	1954	2448	2009
5	1	1083	1585	1625	1780	1625
	32	1131	2011	2130	2784	2011
	128	1144	1682	1883	2383	1881

Table 7: Performance speedup for various optimization for multi-kernel implementation

*It was not possible to put entire HxW i/p matrix for layer1 in shared memory, so in Mk-sw-mt-si-c we coalesced the global memory accesses for layer 1 which was not done in earlier multi-kernel optimization like Mk, Mk-sw-mt and measured the speedup

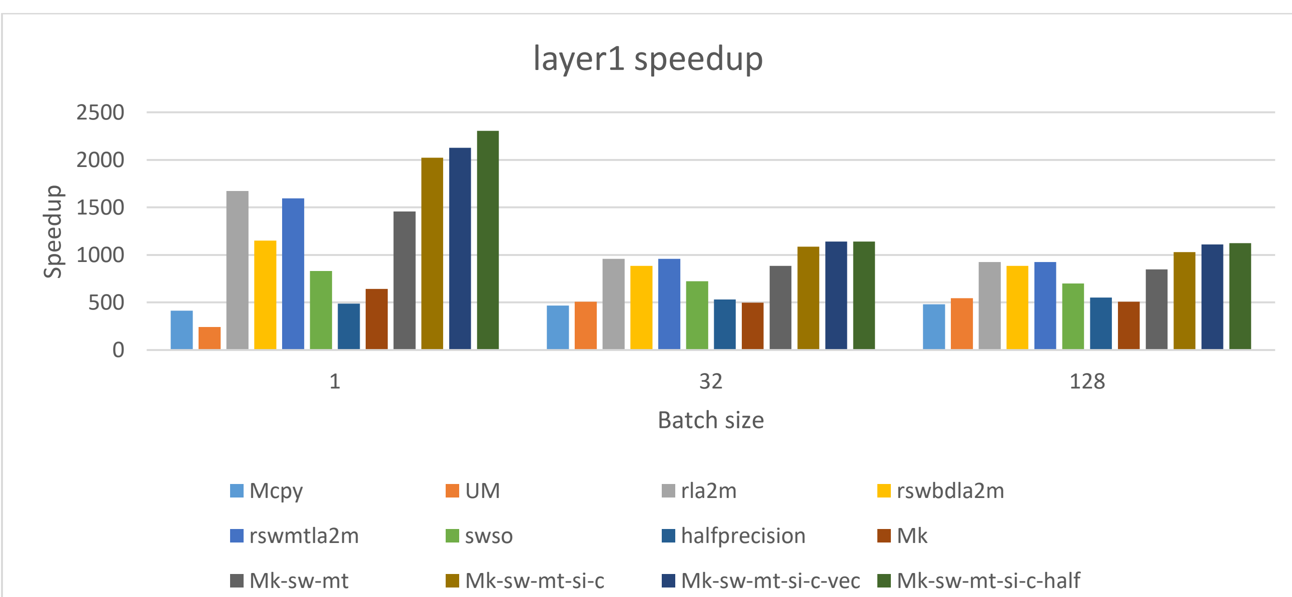


Fig 3a: Speedup of layer 1 for different optimizations

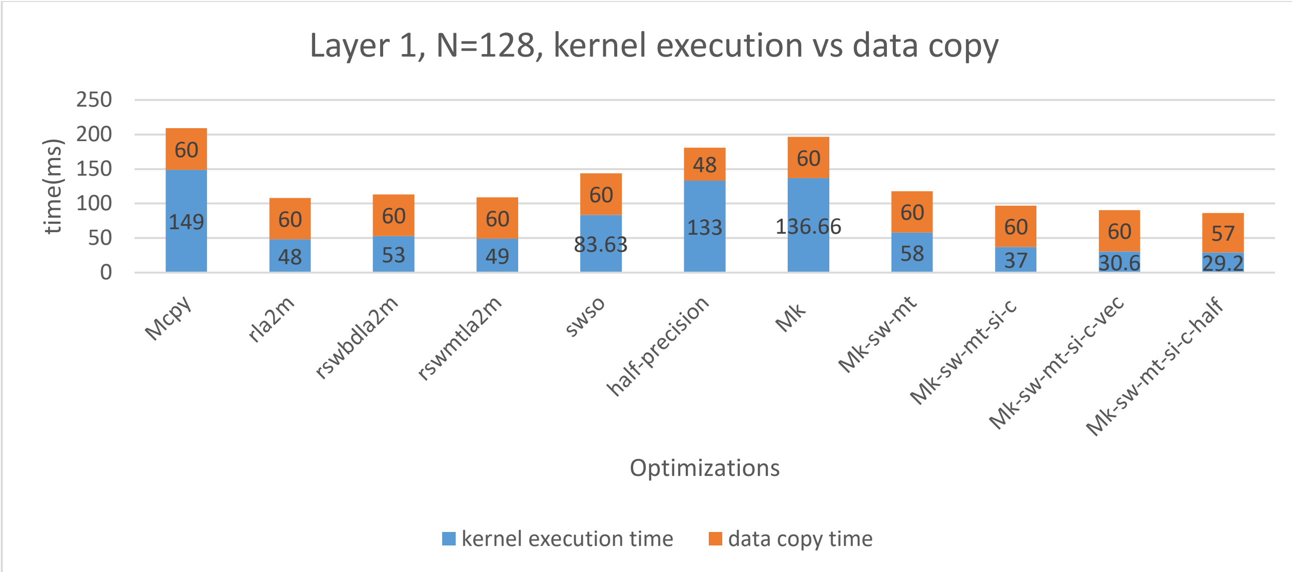


Fig 3b: Breakdown of layer 1 for different optimizations, N=128

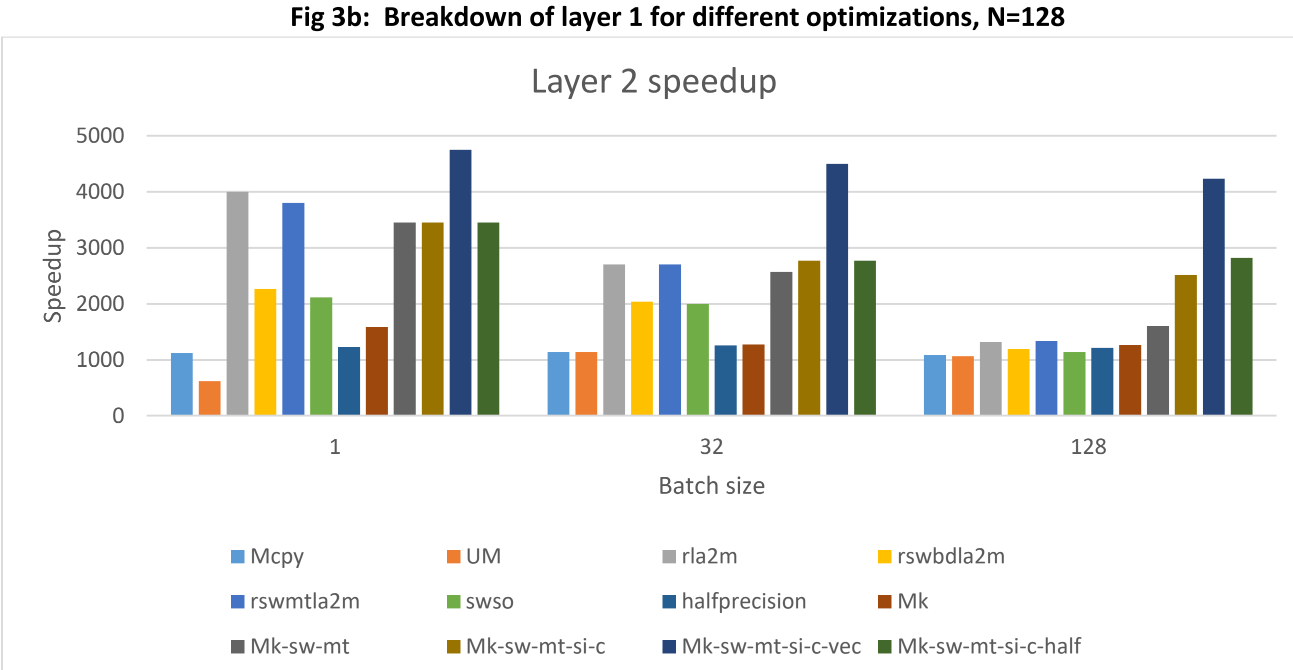


Fig 4a: Speedup of layer 2 for different optimizations

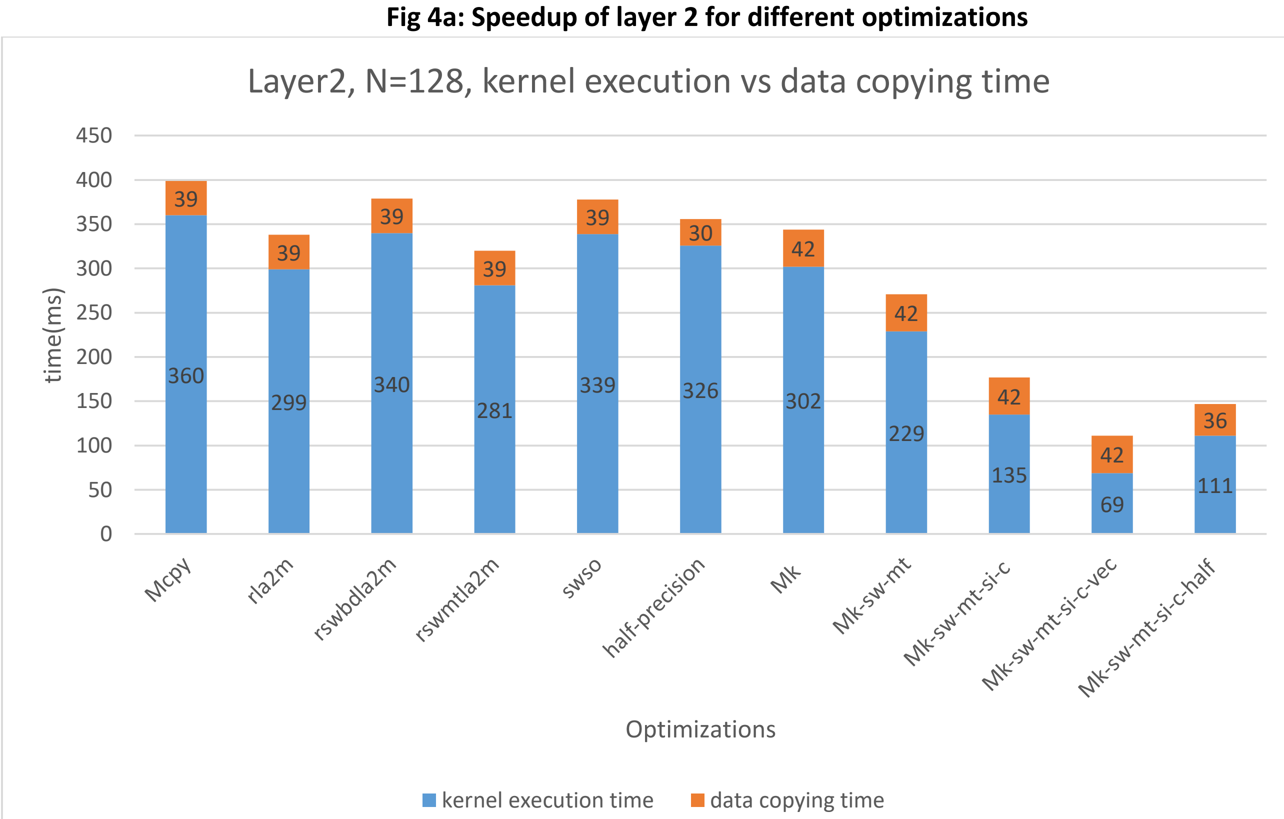


Fig 4b: Breakdown of layer 2 for different optimizations, N=128

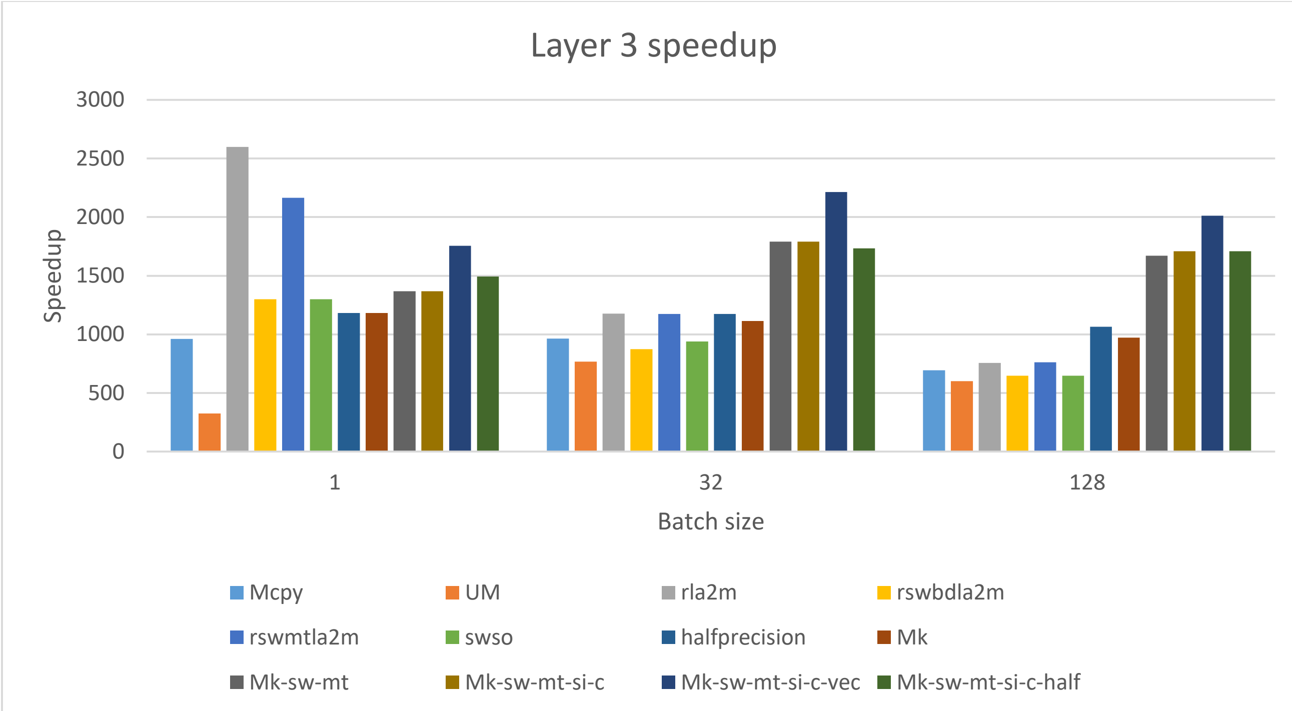


Fig 5a: Speedup of layer 3 for different optimizations

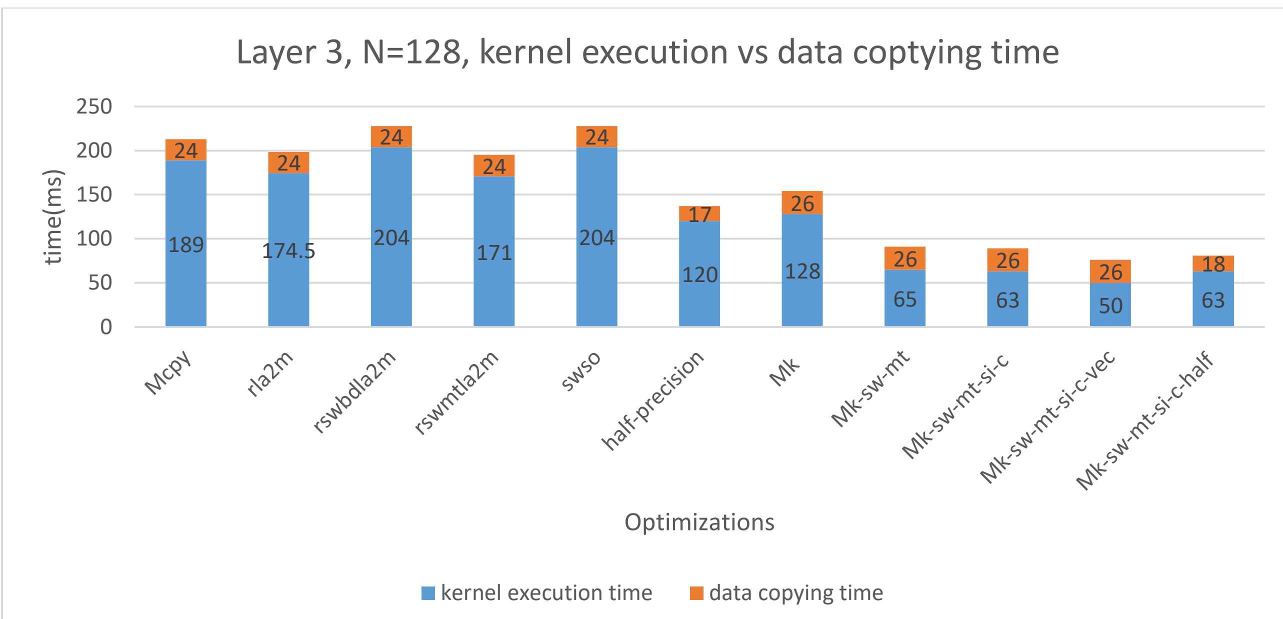


Fig 5b: Breakdown of layer 3 for different optimizations, N=128

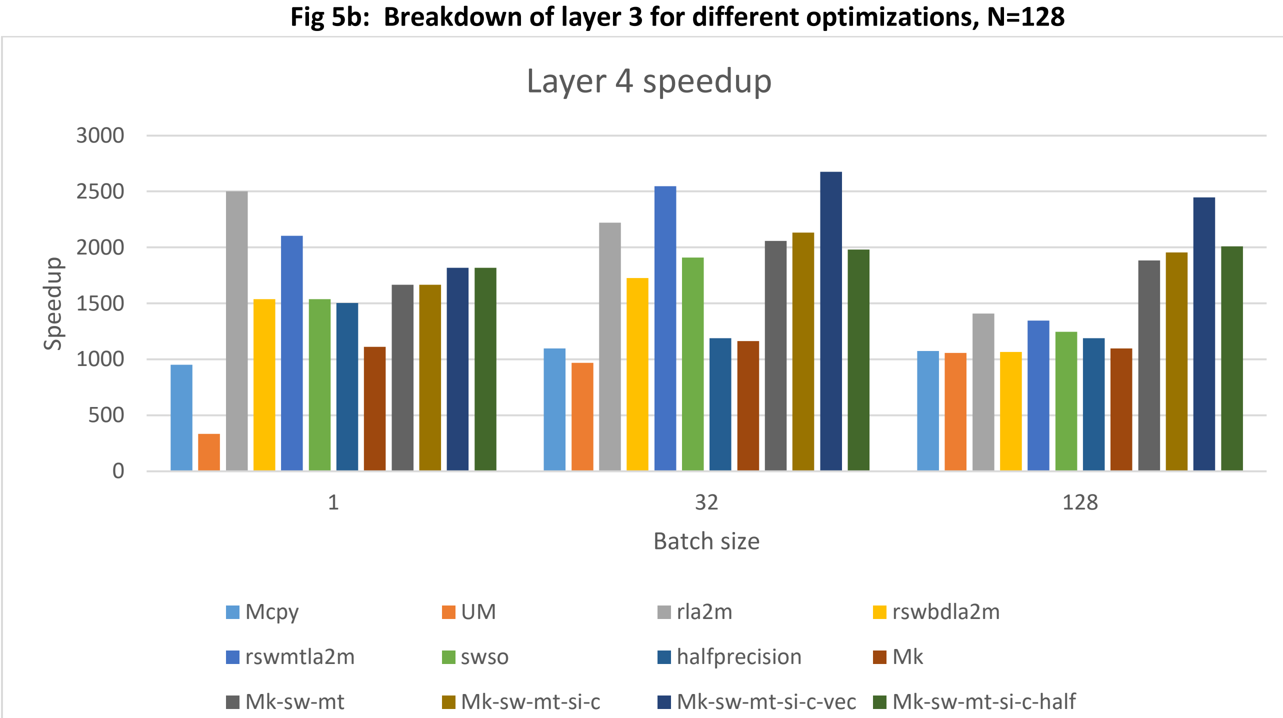


Fig 6a: Speedup of layer 4 for different optimizations

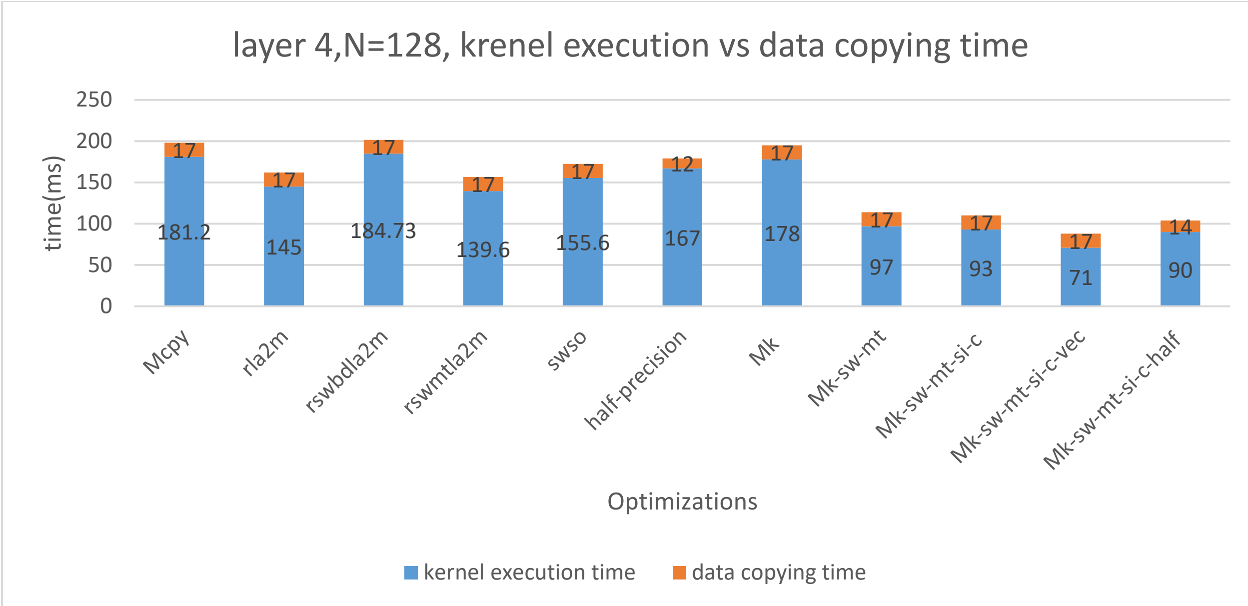


Fig 6b: Breakdown of layer 4 for different optimizations, N=128

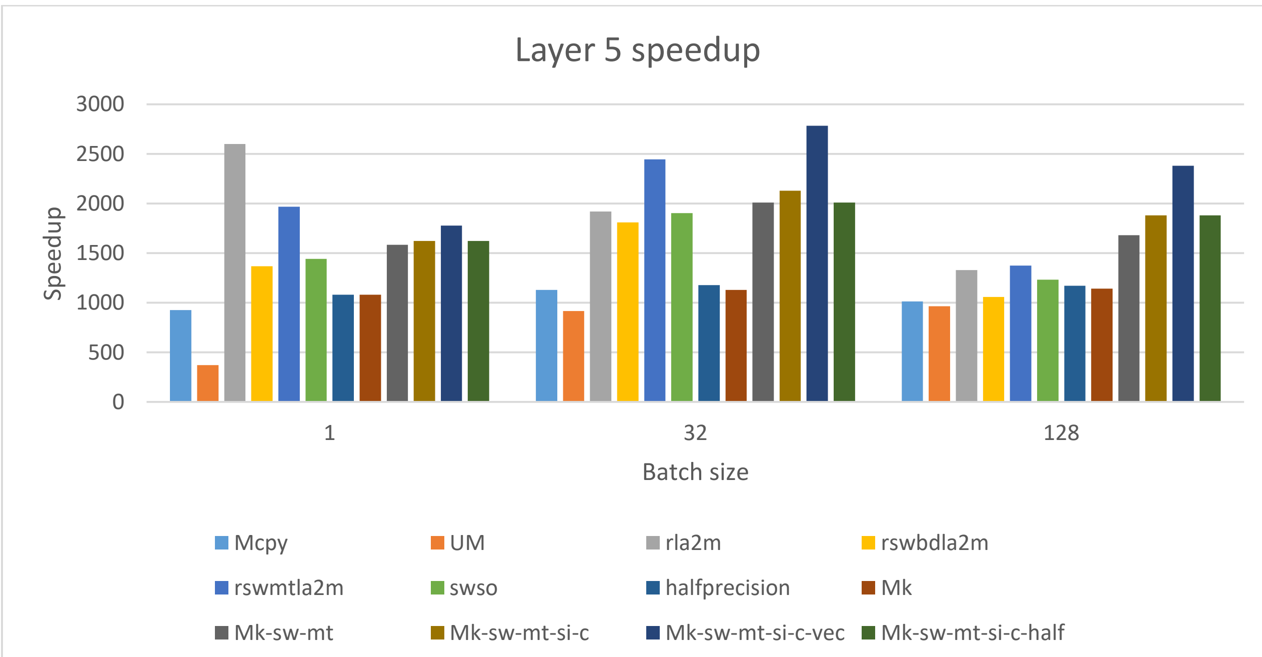


Fig 7a: Speedup of layer 5 for different optimizations

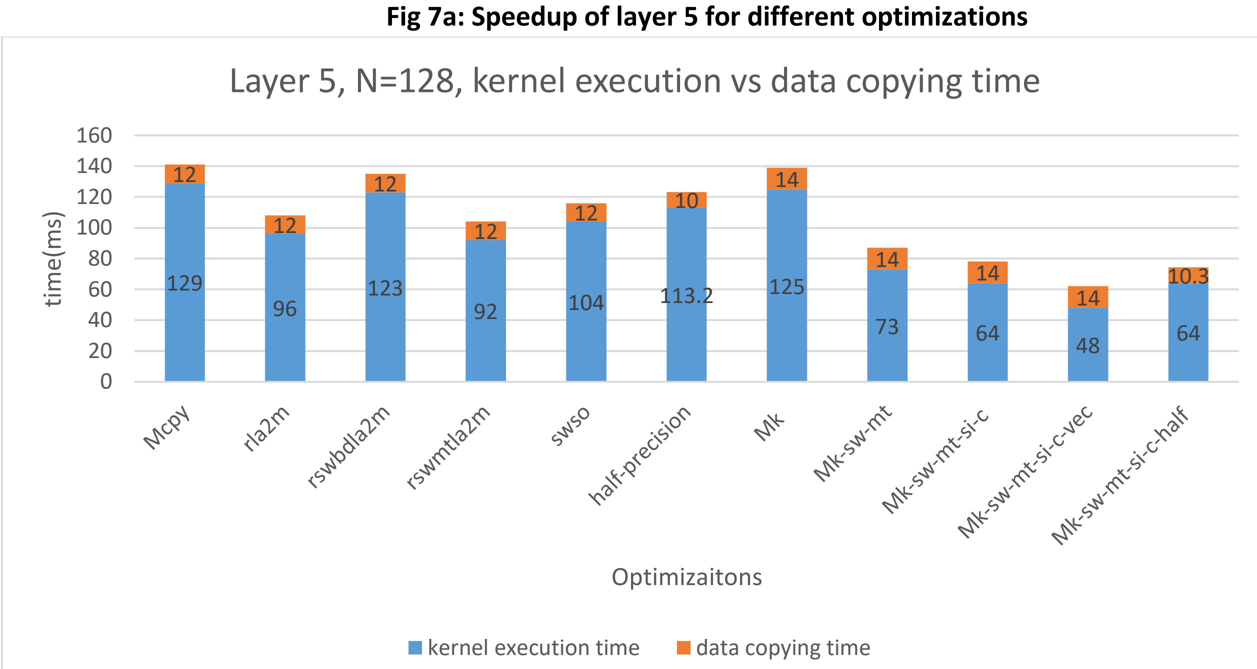


Fig 7b: Breakdown of layer 5 for different optimizations, N=128

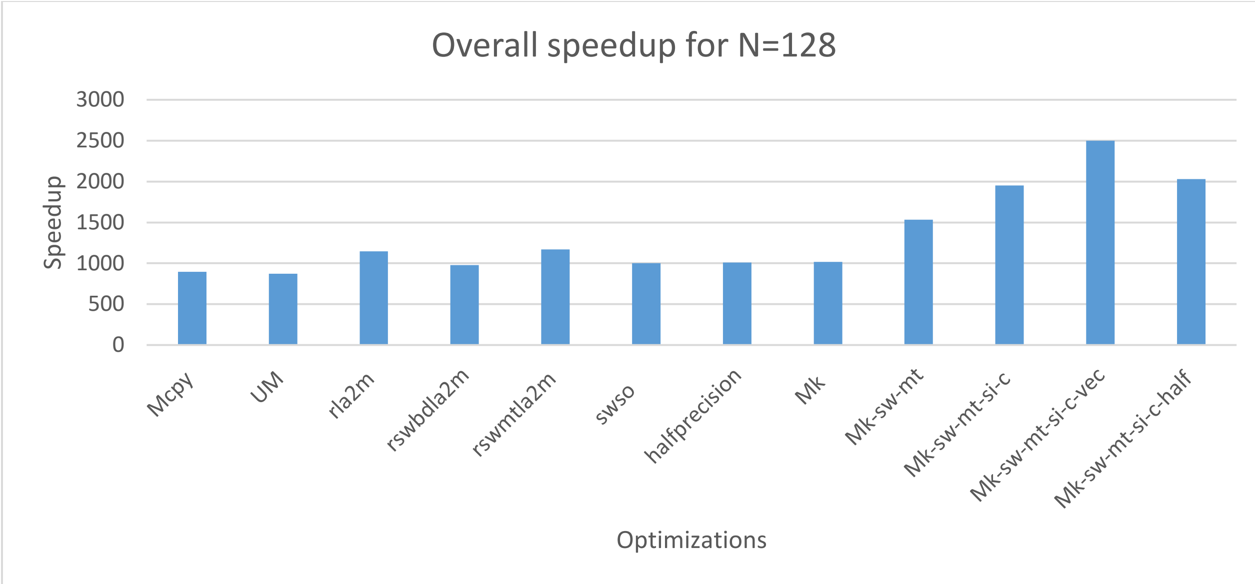


Fig 8a: Overall speedup of all layers for different optimizations,N=128

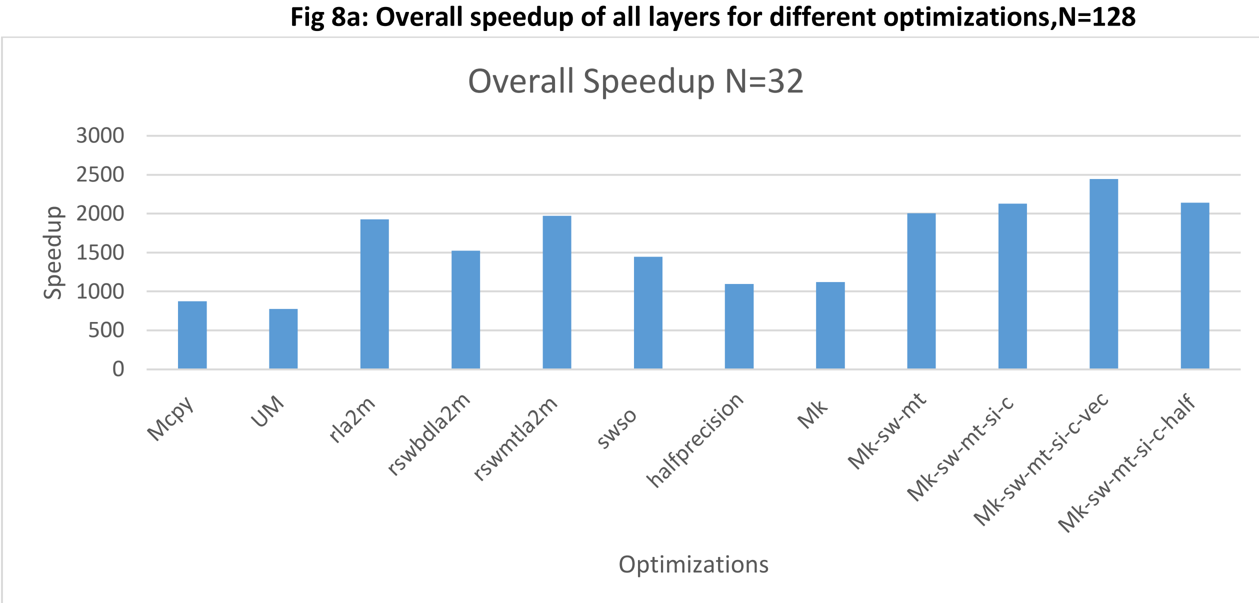


Fig 8b: Overall speedup of all layers for different optimizations,N=32

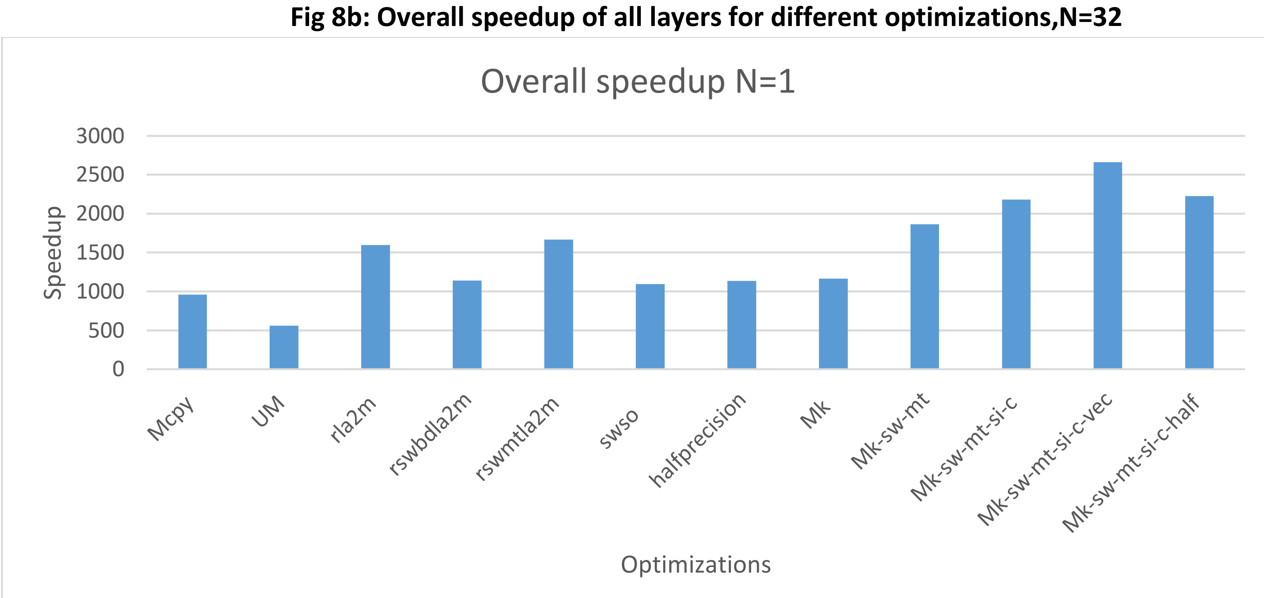


Fig 8c: Overall speedup of all layers for different optimizations,N=128

Optimization name	Batch size	Non-Coalesced speedup (non-coalesced optimization-name)	Coalesced speedup (coalesced optimization-name)
rla2m	1	830(rla2m)	1673(rla2m)
	32	748(rla2m)	958(rla2m)
	128	724(rla2m)	925(rla2m)
rswbdla2m	1	564(rswbdla2m)	1152(rswbdla2m)
	32	255(rswbdla2m)	887(rswbdla2m)
	128	200(rswbdla2m)	885(rswbdla2m)
rswmtla2m	1	708(rswmtla2m)	1596(rswmtla2m)
	32	249(rswmtla2m)	958(rswmtla2m)
	128	195(rswmtla2m)	925(rswmtla2m)
swso	1	446(swso)	830(swso)
	32	240(swso)	1041(swso)
	128	191(swso)	700(swso)
Mk-sw-mt	1	1456(Mk-sw-mt)	2024(Mk-sw-mt-si-c)

	32	887(Mk-sw-mt)	1088(Mk-sw-mt-si-c)
	128	847(Mk-sw-mt)	1030(Mk-sw-mt-si-c)

Table 8: Comparison of speedup for coalesced and non-coalesced access for layer 1

layer	kernel time (best-case)(ms)	data copying time(ms)
1	30.6	60
2	69	42
3	50	26
4	71	17
5	48	14

Table 9: Comparison of data copying and execution time for Mk-sw-mt-si-c-vec optimization for N=128

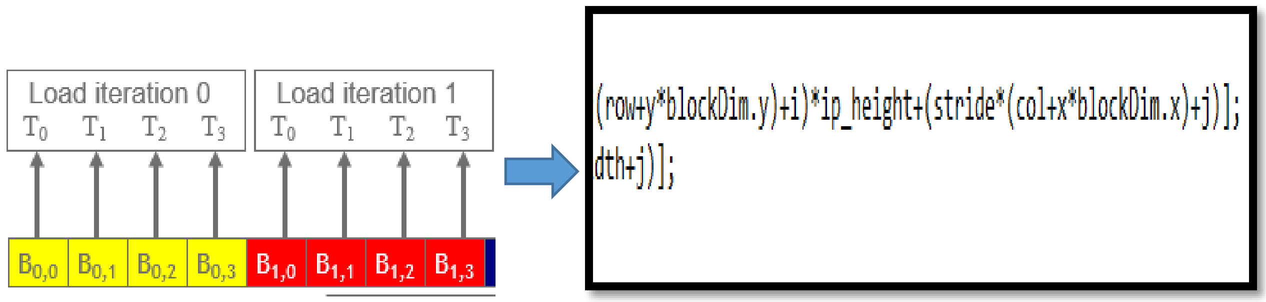


Fig 9: Code snippet and example of coalesced memory access in layer 1

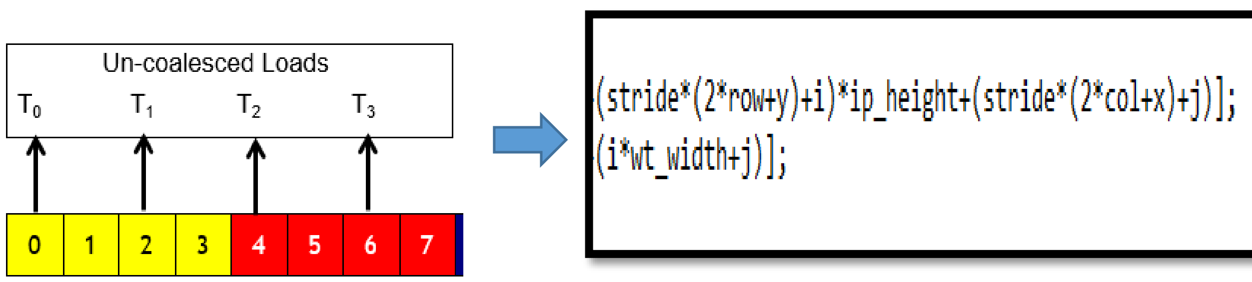


Fig 10: Code snippet and example of non-coalesced memory access in layer 1

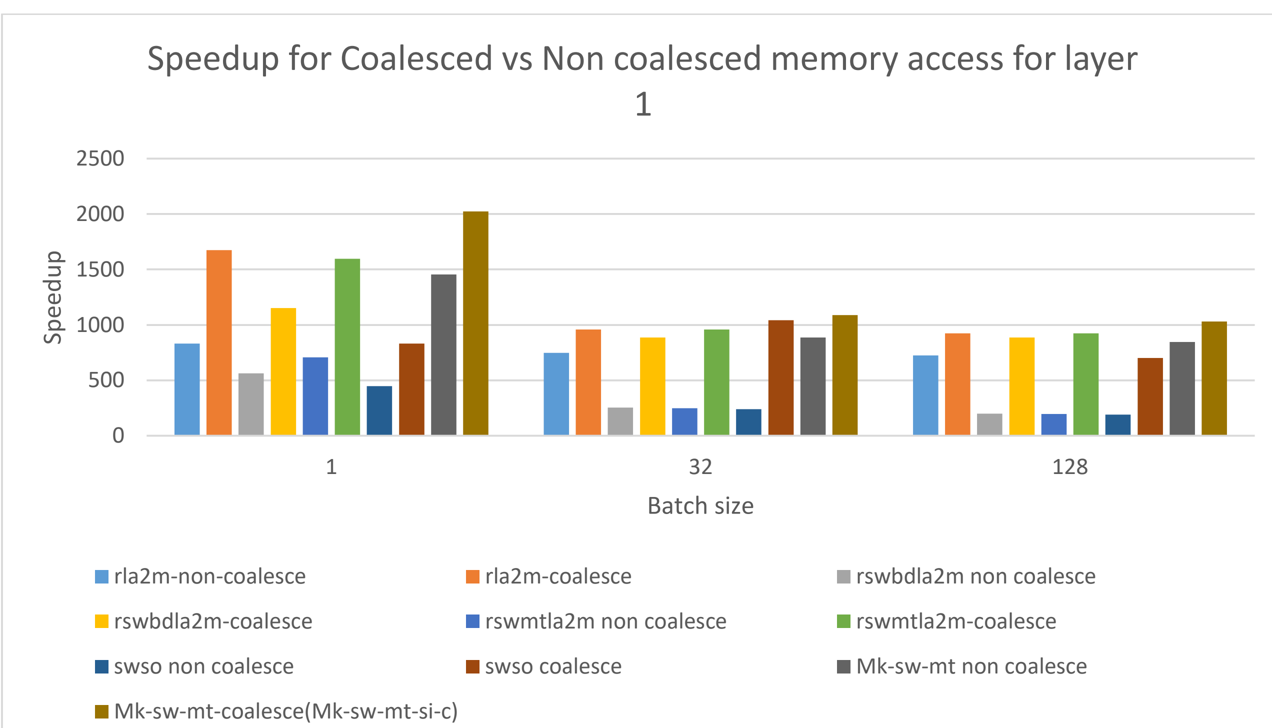


Fig 11: Performance comparison for coalesced and non-coalesced memory access in layer 1

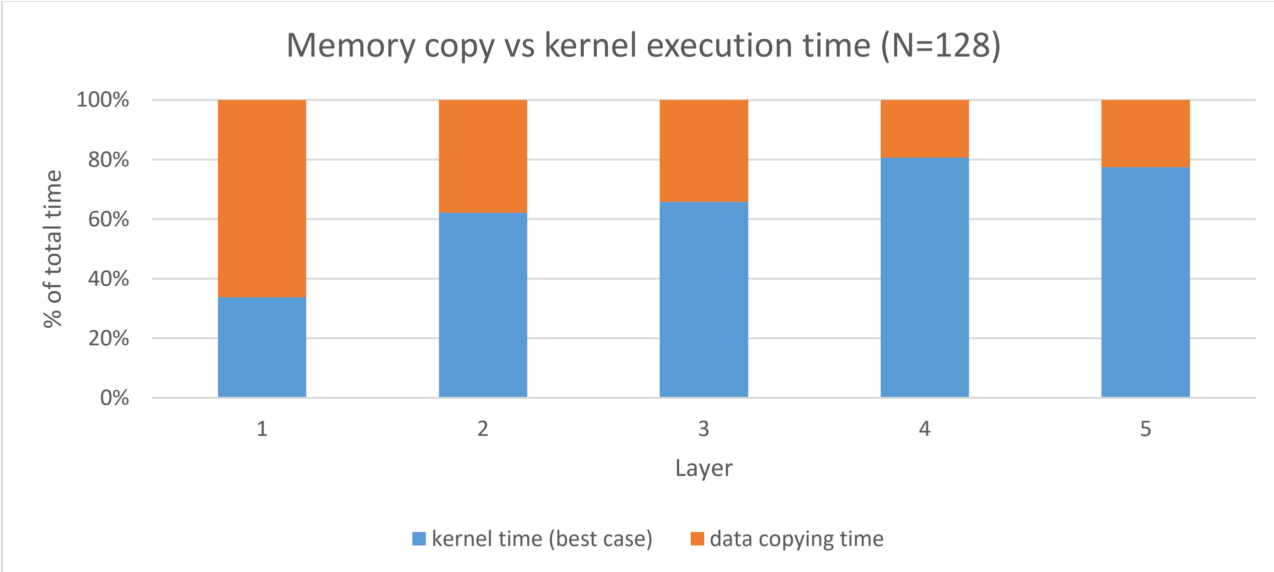


Fig 12a: Comparison of memcpy time and kernel execution time for all layers(N=128) for Mk-sw-mt-si-c-vec optimization

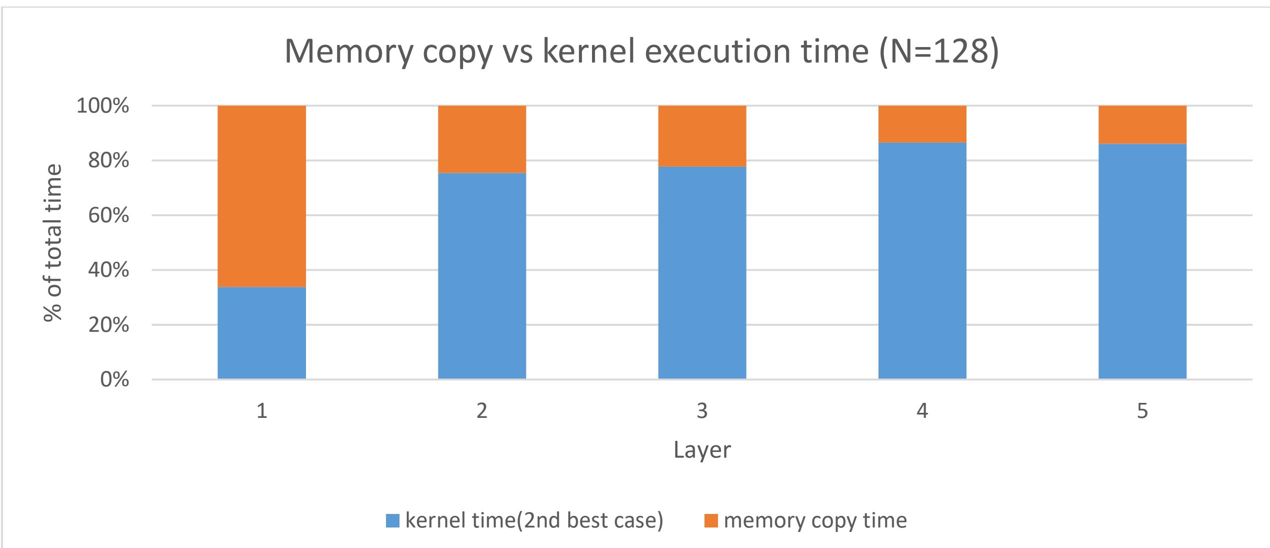


Fig 12b: Comparison of memcpy time and kernel execution time for all layers(N=128) for Mk-sw-mt-si-c-half optimization

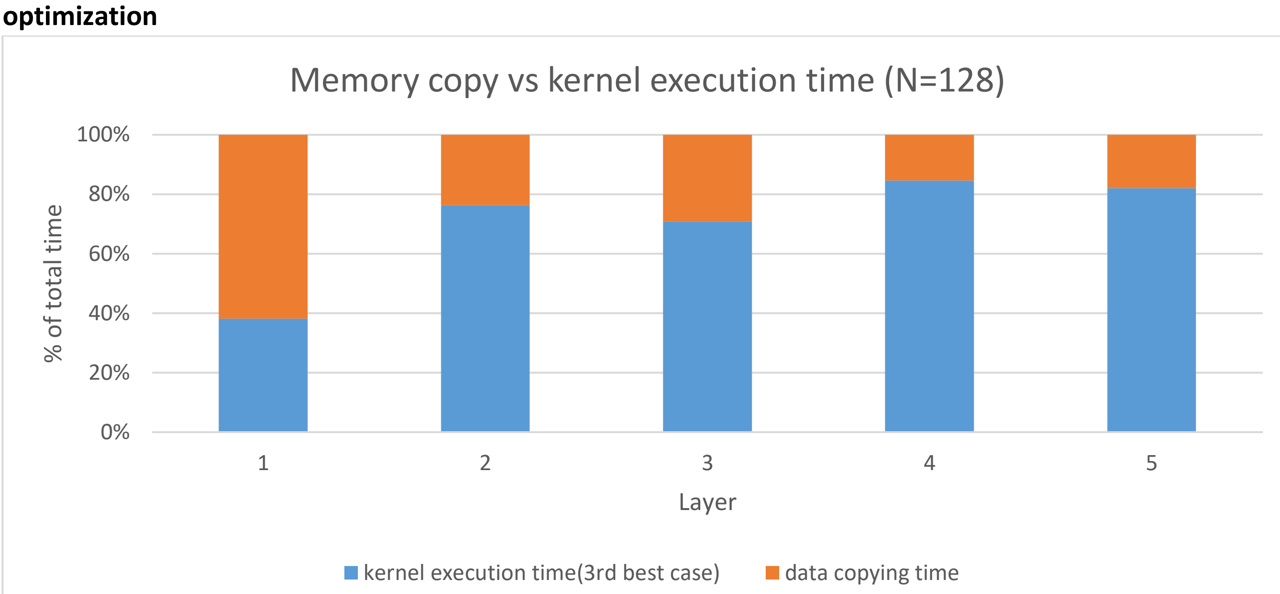


Fig 12c: Comparison of memcpy time and kernel execution time for all layers(N=128) for Mk-sw-mt-si-c optimization

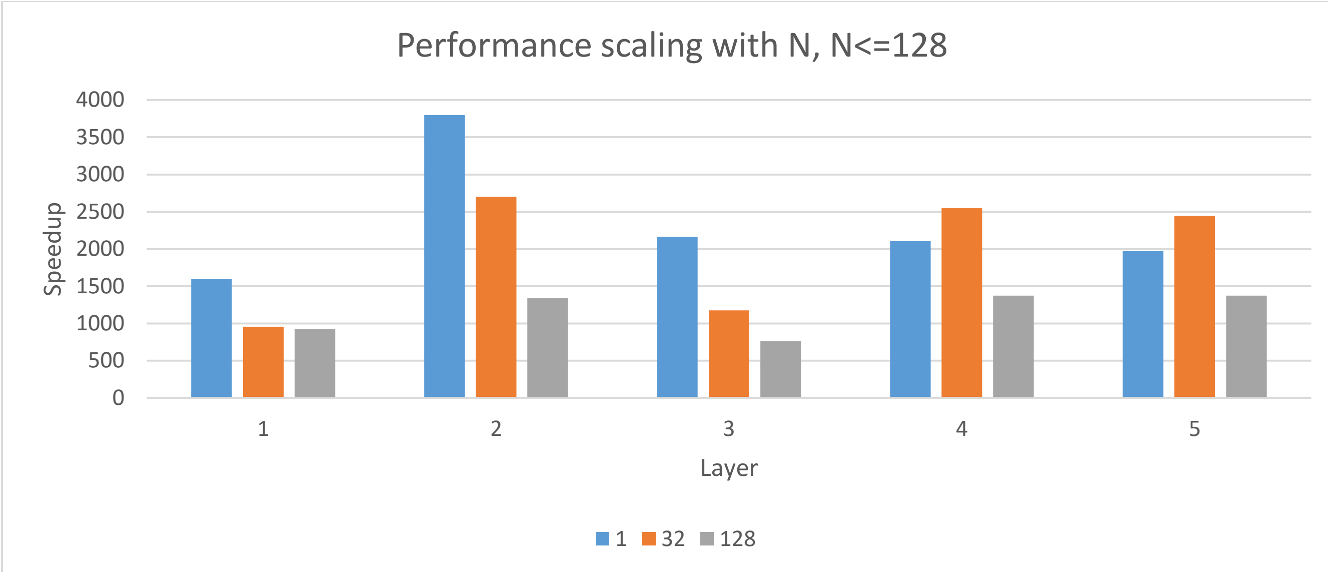


Fig 13a: Effect of scaling on speedup for rswmtla2m optimization for N=1,32,128

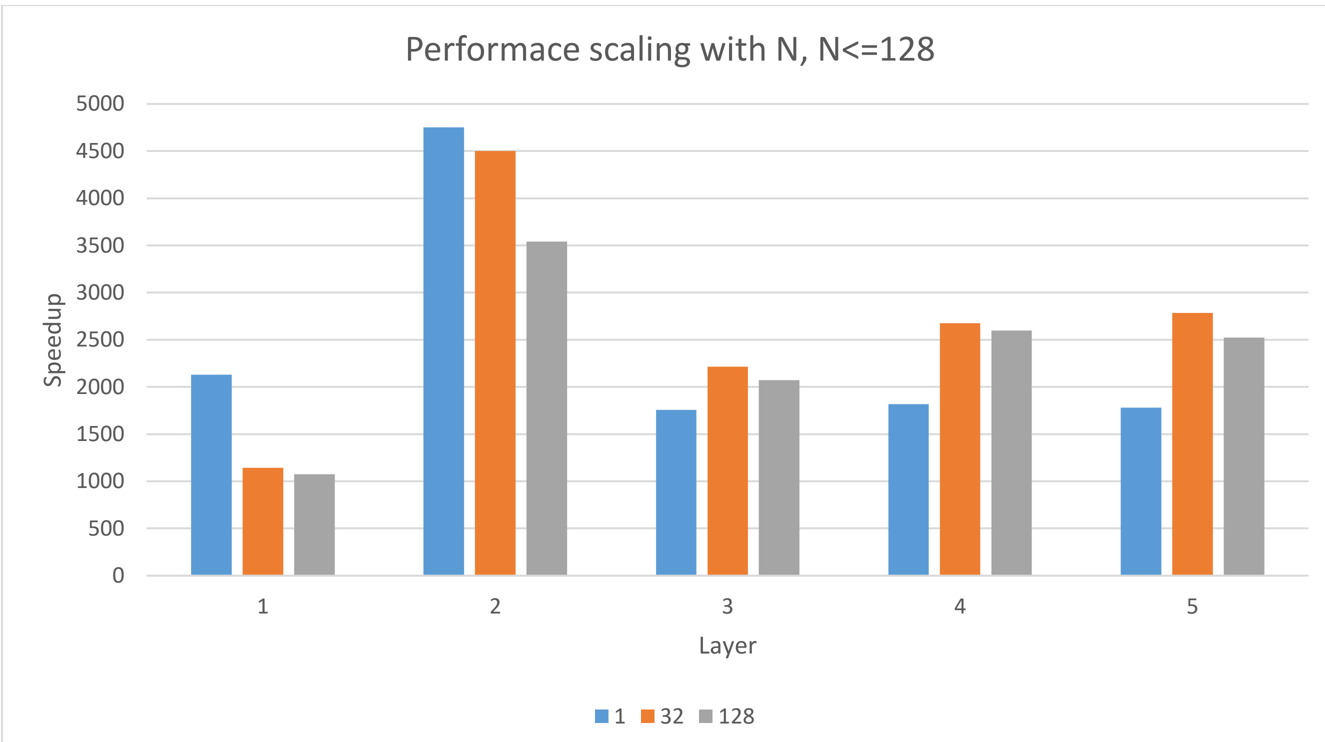


Fig 13b: Effect of scaling on speedup on Mk-sw-mt-si-c-vec optimization

Layer	Batch size	CPU time(s)	Kernel exec time (ms)	Layer	Batch size	CPU time(s)	Kernel exec time (ms)
1	1	0.83	0.4	4	1	2	0.4
	32	23.95	11.3		32	53.5	17.8
	128	100	53.3		128	213	139.6
	256	187	113		256	420	308
	512	375	209		512	837	638
	1024	749	404		1024	1671	1277
	2048	1491	809		2048	3340	2584
2	1	3.8	0.8	5	1	1.3	0.33
	32	108	31.32		32	36.2	11.86
	128	432	281		128	143	92
	256	830	599		256	280.4	205
	512	1657	1199		512	558	425
	1024	3310	2398		1024	1113	851
	2048	6623	4795		2048	2227	1703
3	1	1.3	0.3				
	32	37.63	27.4				
	128	147	171				
	256	284	364				
	512	562	749				
	1024	1118	1514				
	2048	2232	3028				

Table 10: Kernel execution time with device scaling for rswmtla2m optimization

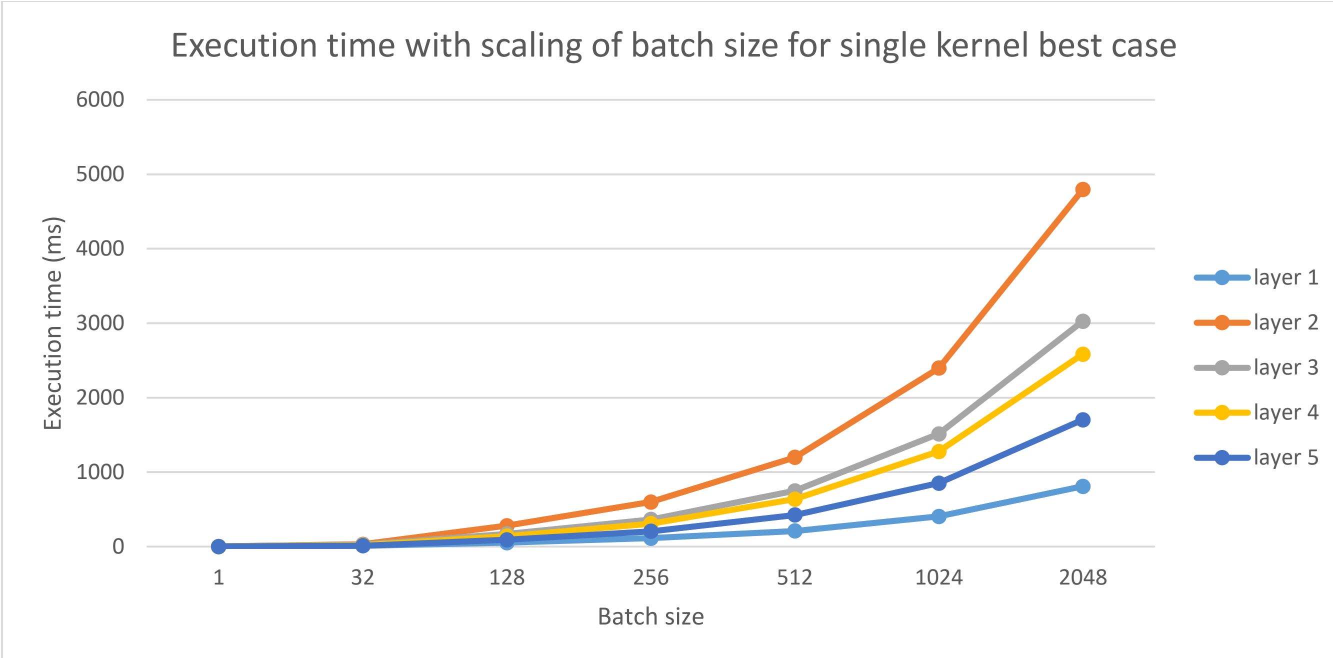


Fig 14a: Effect of scaling on kernel execution time for rswmtla2m optimization

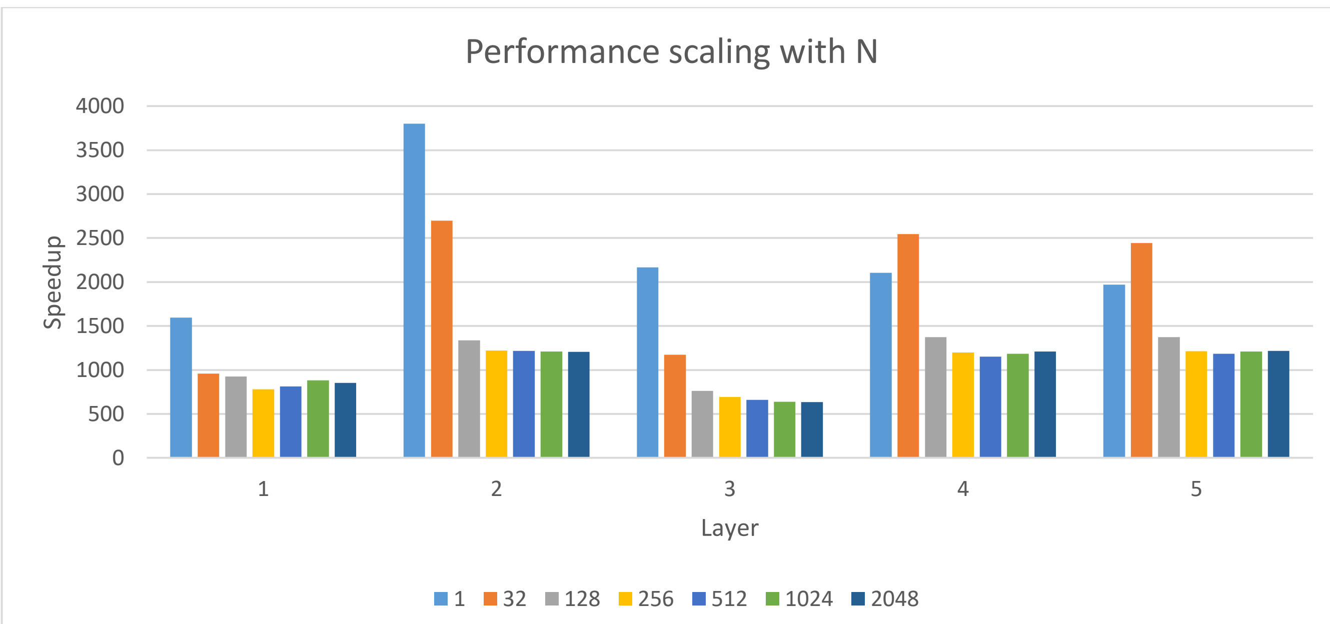


Fig 14b: Effect of scaling on speedup for rswmtla2m optimization

Layer	Batch size	CPU time(s)	Execution time(ms)	Layer	Batch size	CPU time(s)	Execution time (ms)
1	1	0.83	0.7	4	1	2	0.54
	32	23.95	7.8		32	53.5	16
	128	100	33		128	213	66.3
	256	187	62		256	420	128.65
	512	375	120.55		512	837	240.98
	1024	749	234.83		1024	1671	467.06
	2048	1491	472.21		2048	3340	933
2	1	3.8	0.56	5	1	1.3	0.38
	32	108	15		32	36.2	10
	128	432	79.11		128	143	43.98
	256	830	153.4		256	280.4	87.93
	512	1657	299.15		512	558	160
	1024	3310	626.56		1024	1113	315
	2048	6623	1264		2048	2227	625.51
3	1	1.3	0.37				
	32	37.63	11				
	128	147	46.6				
	256	284	95.06				
	512	562	171				
	1024	1118	416				
	2048	2232	1346				

Table 11: Kernel execution time with device scaling for exp-Mk-sw-mt-si-c-vec, single-kernel optimization

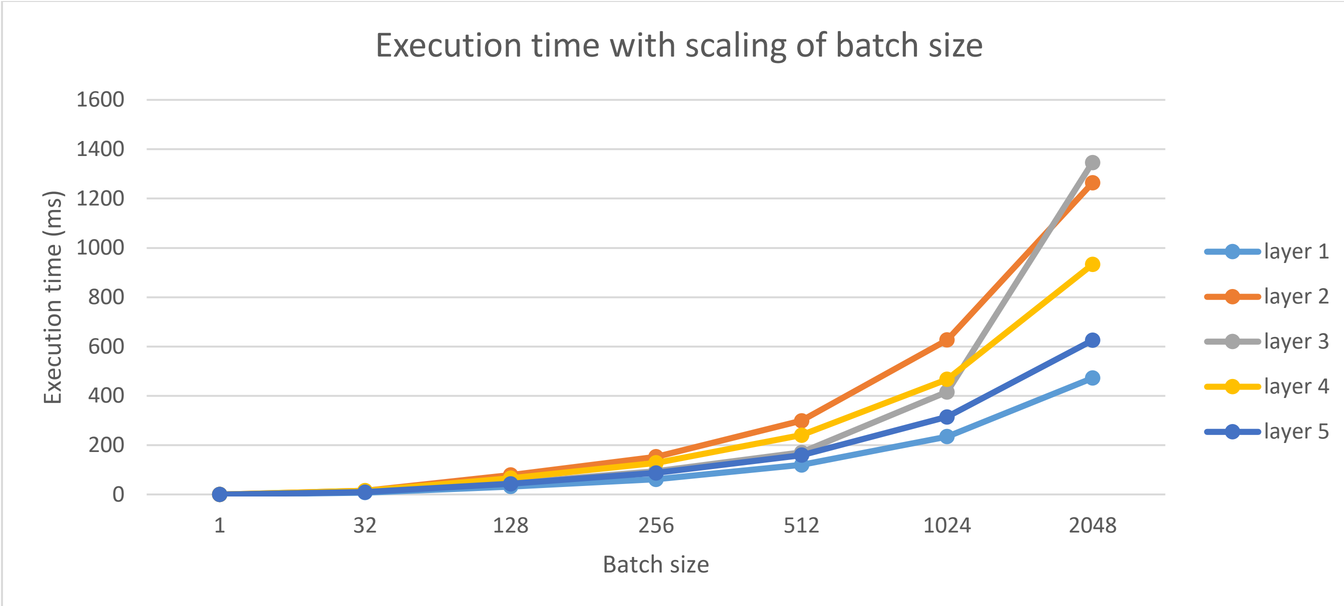


Fig 15a: Effect of scaling on execution time on exp-Mk-sw-mt-si-c-vec (single kernel, less precise) optimization

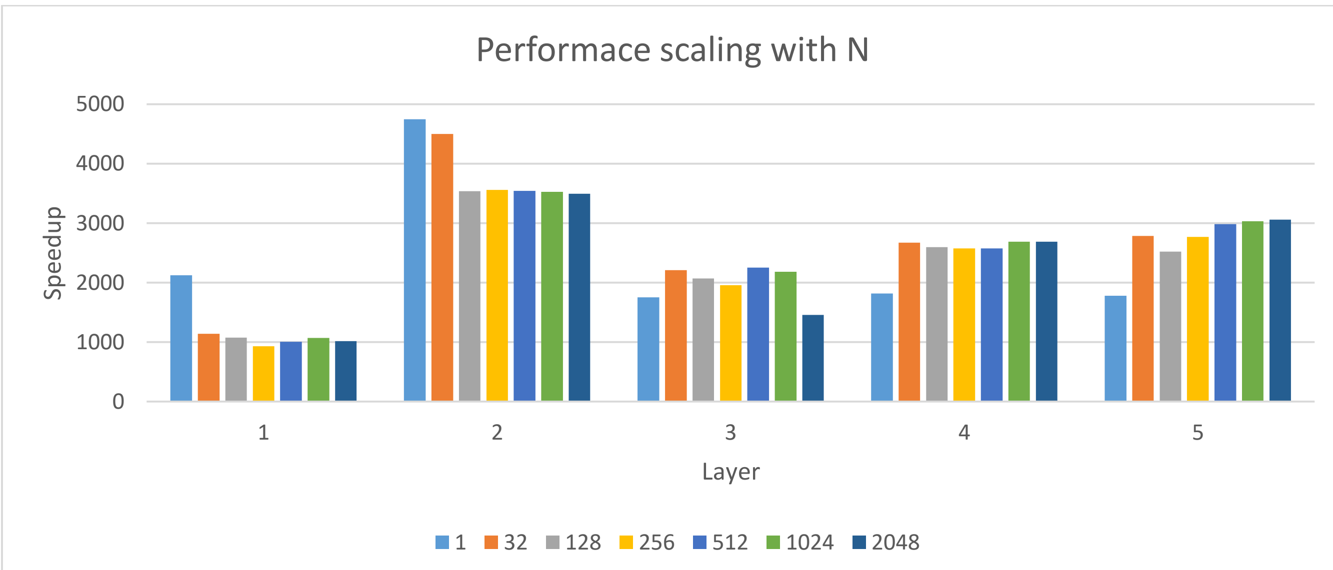


Fig 15b: Effect of scaling on speedup for exp-Mk-sw-mt-si-c-vec(single kernel , less precise) optimization.

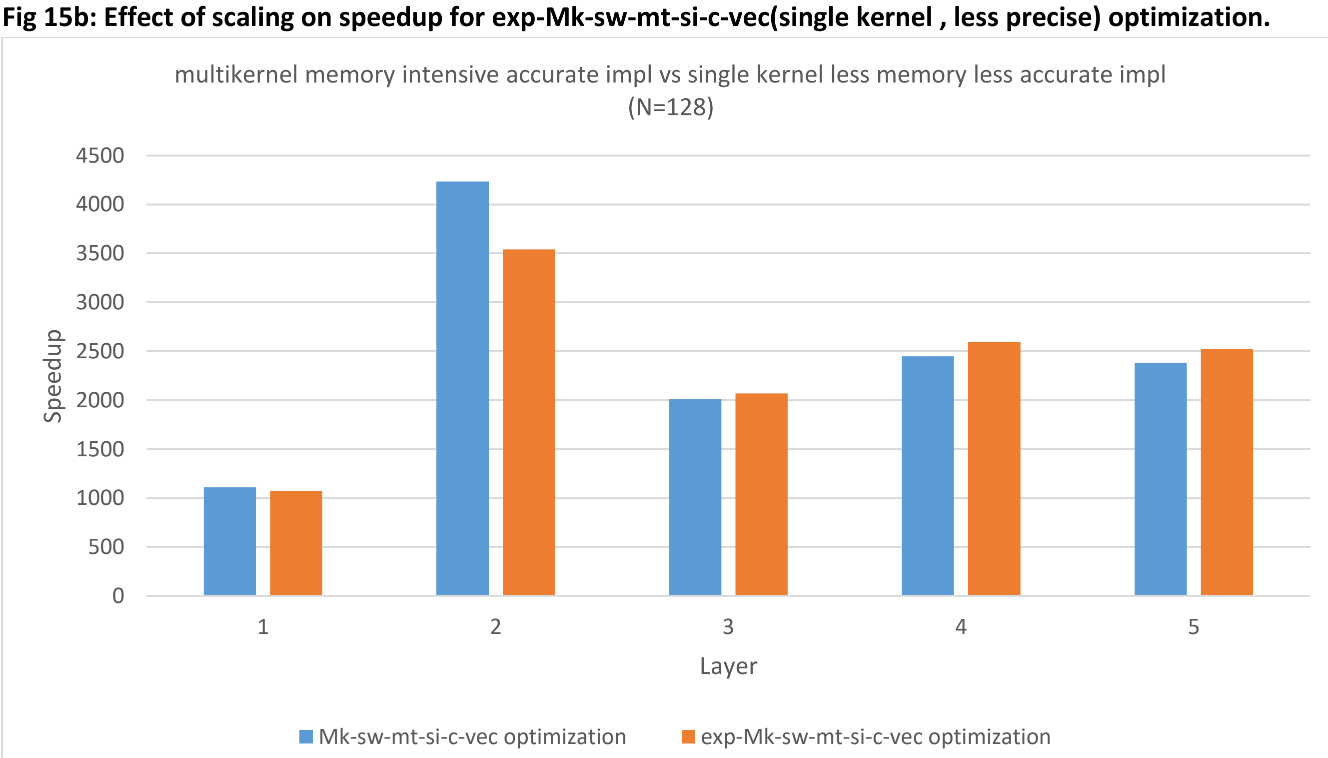


Fig 16 : Speedup comparison of Mk-sw-mt-si-c-vec vs exp-Mk-sw-mt-si-c-vec(single kernel, less precise) optimization for N=128

Reference:

1. https://piazza.com/purdue/spring2020/ece69500009/resources, 11-mlandaccelerators.pptx
2. https://medium.com/@prrama/an-introduction-to-writing-fp16-code-for-nvidias-gpus-da8ac000c17f
3. https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html
4. https://piazza.com/purdue/spring2020/ece69500009/resources, 08-shmemcudaend.pptx
5. https://piazza.com/purdue/spring2020/ece69500009/resources, 07-dramcoalescingthreadmapping.pptx
6. https://piazza.com/purdue/spring2020/ece69500009/resources, 06-mmoptocodedram.pptx
7. https://piazza.com/purdue/spring2020/ece69500009/resources, 05-matmulopto.pptx
8. https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/
9. https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/