AVX Optimizations:
VSIPL Benchmarks.
Report 3.

N.*A. Software Ltd.*

21st February 2011.

Table of Contents

1. Introduction.

In April 2010 NA Software Ltd (NAS) produced a report into the performance of the Vector Signal Processing Library (VSIPL: www.vsipl.org) on Intel's AVX hardware. The report compared the SSE performance of the following operations to the equivalent AVX performance. To do this we used our own DSP algorithms and the Integrated Performance Primitives (IPP) Library under a series of VSIPL wrappers. The table below shows the VSIPL operations and the benchmark parameters that this report studied as given in the initial statement work.

Table 1: VSIPL operations.

Routine	Benchmark Parameters.
1D FFT, complex-complex, in-place	256, 1K, 4K, 16K, 256K, 512K points
2D FFT, complex-complex, in-place.	256*256, 1K*100, 4K*50, 16K*20, 64K*20, 128K*20 64*64, 128*128, 256*256, 512*512, 1K*1K, 2K*2K [points*rows]
Multiple FFT, complex-complex, in-place.	256*256, 1K*100, 4K*50, 16K*20, 64K*20, 128K*20 64*64, 128*128, 256*256, 512*512, 1K*1K, 2K*2K [points*rows]
Complex matrix transpose.	256*256, 1K*100, 4K*50, 16K*20, 64K*20
Complex vector multiply.	256, 1K, 4K, 16K, 32K, 64K, 128K
Vector sine.	256, 1K, 4K, 16K, 32K, 64K, 128K
Vector cosine.	256, 1K, 4K, 16K, 32K, 64K, 128K
Vector square root.	256, 1K, 4K, 16K, 32K, 64K, 128K
Vector scatter.	256, 1K, 4K, 16K, 32K, 64K, 128K
Vector gather.	256, 1K, 4K, 16K, 32K, 64K, 128K

All the above lengths are given in complex cells. In November 2010, NA Software was asked to review the achieved performance on the latest AVX platform. This was close to AVX platform being released for sale to the general public and formed our report 2 on AVX. It described the difference in performance of SSE and AVX versions with the above operations as VSIPL calls. Version 1 of the report used an early release of the AVX platform and version 6 of the IPP library. Report 2 used the latest AVX platform and version 7.0.205.07 of the IPP library which had been substantially improved for AVX.

Report 1 and 2 compared SSE and AVX optimized code running on the same AVX platform. In this report (Report 3) we also compare the SSE programs running on an Intel SSE4 Arrandale platform to the AVX optimized code running on the AVX platform. Report 1 and 2 highlighted the extra processing power of the AVX platform. This report shows the extra processing power but also the substantially higher memory speed and memory performance of the AVX machine.

This report is organized into the following sections:

Section 2: Routine development – states how the algorithms and testing code were developed along with version numbers of tools and libraries that have been used to perform the study.

Section 3: The SSE and AVX performance – compares SSE optimized code to AVX optimized code on the same AVX platform. The AVX hardware is fully backward compatible to SSE. Therefore, SSE optimized code will run on any AVX machine. The performance gain noted in this section is due to the AVX optimized code using the larger richer AVX instruction set along with the full 256 bits of the platform registers. This shows the performance gain of the wider registers combined with the greater functionality of the instruction set.

Section 4: The Arrandale SSE and AVX performance – compares SSE optimized code running on a SSE platform to AVX optimized code running on an AVX test platform. When performing this comparison we have used one of the latest SS4 platforms, the Intel Arrandale i5. The performance gain of AVX is much higher in this section than is reported in section 3. This is down to the improved memory controller and two 128 bit load ports of the AVX machine giving it a faster memory speed.

Section 5: The compiler performance – compares Intel's icc compiler and the GNU gcc compiler with c code performing the study operations on the AVX platform.

Section 6: Conclusions – summarizes the results obtained in the sections above.

2. Routine Development.

In order to carry out study 1 NAS developed AVX code for the operations being studied. This code was tested against a trusted SSE version of the VSIPL library and it was verified that it produced the same results. As well as using our own AVX code in this report NAS also studied Intel's AVX optimized IPP library (Version 7). This report describes the performance of both the IPP library and the NAS AVX code embedded within VSIPL, and discusses the best optimized AVX solution for each operation being studied. Once the development code had been optimized and tested then timings were taken of both the SSE version and the AVX version of each test operation. In report 1 and 2 all these timings were produced on the same 64 bit AVX platform running at 2.0GHz under the Linux operating system (Fedora 13). In report 3 a 64 bit Arrandale SS4 platform is also used with the equivalent operating frequency and the same operating system. This study uses version 7.0.205.07 of the IPP library which has been fully optimized for AVX. Initially, both the SSE and AVX timing programs were compiled with gcc version 4.4.5 and the appropriate AVX flags. This report then goes on to compare the performance of using Intel's icc compiler to using the gcc compiler. To do this version 12 of the icc compiler has been used.

3. The SSE and AVX Performance.

3.1. 1D in-place complex-to-complex FFT.

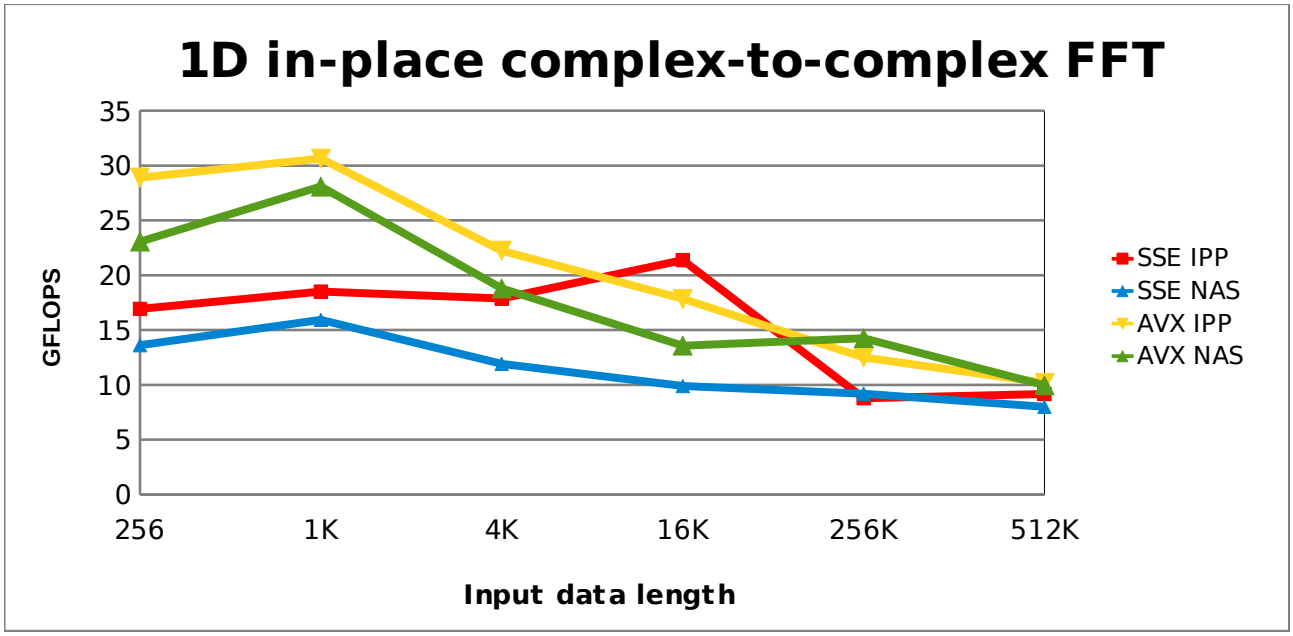
In VSIPL the in-place complex-to-complex FFT operation is carried out by a call to the library function vsip_ccfftip_f, after views and FFT plans have been created. Thavx_report4.odte following table and graph show the obtained timings when the SSE and AVX timing programs were executed. We give here both NAS based timings and timings based on Intel's IPP library. The results in the table are in microseconds and the results in the graph are presented in GFlops.

Table 2: vsip_ccfftip_f timings.

Input Data Length	256	1K	4K	16K	256K	512K
SSE IPP	0.61	2.77	13.75	53.6	2,688	5,438
SSE NAS	0.75	3.22	20.64	116.1	2,571	6,237
AVX IPP	0.35	1.67	11.05	64.3	1,888	4,890
AVX NAS	0.44	1.82	13.08	84.5	1,656	4,999

Figure 1: complex to complex 1D in-place FFT performance.

MFLOPS = 5 N Log_2(N) / (time for one FFT in microseconds)



The blue and green graphs show the performance of the NAS 1D FFT with the blue graph being SSE version and the green graph being the AVX performance. This shows a substantially improved performance across all data lengths. The red and yellow graphs show the performance of the IPP library with the red graph being the SSE graph. In the last report we used version 6 of the IPP library which shows no performance improvement for AVX. You can see above that the 1D FFT performance of the IPP Library in version 7 is extremely good. The performance is substantially better than SSE until we get to a data length of 8K. The results above show that IPP has the best performance with data lengths less than 256K. However, both libraries highlight the advantage of using an AVX optimized solution. At a data length of 1K IPP has an AVX speed up of 79% and NAS has a speed up of 87%. These percentages are based on 100% representing an AVX time of half the SSE time.

3.2. 2D in-place complex-to-complex FFT.

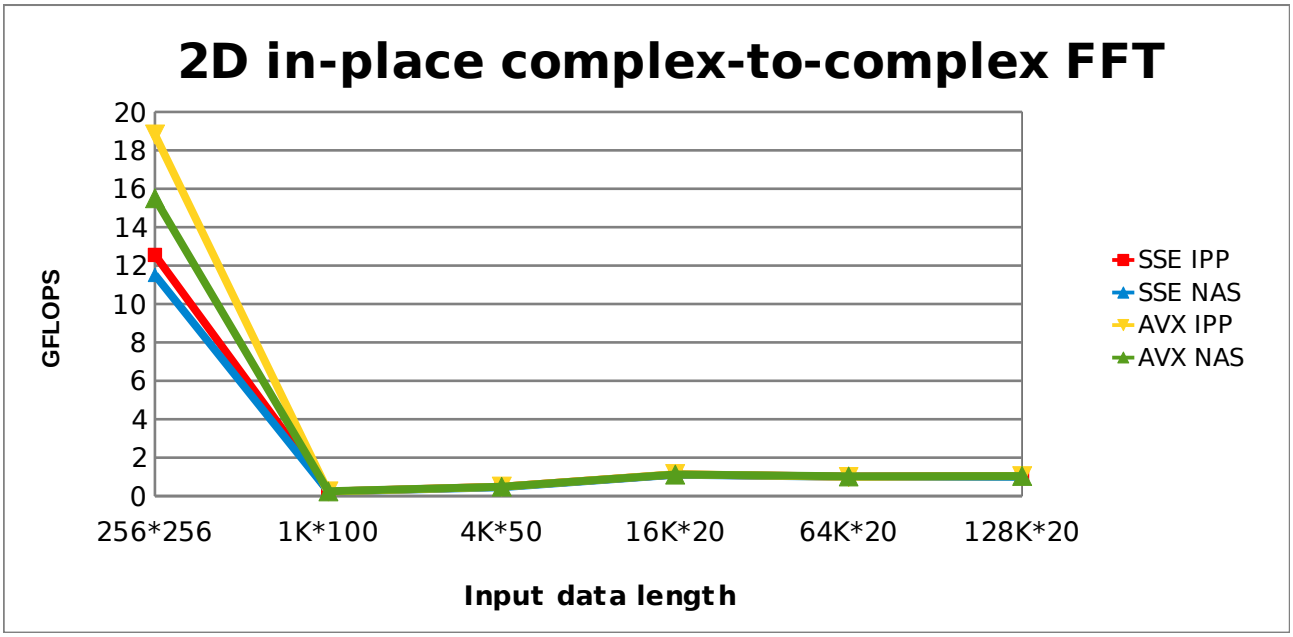
The 2D in-place complex-to-complex operation is carried out by the library call vsip_ccfft2dip_f. This function gets called after VSIPL data matrix views and FFT plans have been created. The following table and graph show the timing results of running the set benchmark parameters. The results in the table are in microseconds and the results in the graph are in GFlops.

Table 3: vsip_ccfft2dip_f timings.

Input Data Length	256*256	1K*100	4K*50	16K*20	64K*20	128K*20
SSE IPP	418	35,534	37,284	26,721	131,128	267,319
SSE NAS	455	35,578	39,748	27,143	133,968	280,492
AVX IPP	279	35,956	36,928	26,323	133,607	269,949
AVX NAS	338	34,655	37,018	26,824	130,293	267,780

Figure 2: complex to complex 2D in-place FFT performance.

MFLOPS = $5NLog_2(N)*M + 5MLog_2(M)*N$ / (time for one FFT in microseconds)



The above graph shows that the performance improvement for 256*256 for the NAS code is 51% and for IPP based code it is 67%. However, the performance for the other lengths is not significantly improved. One reason for this is that the data sizes do not produce aligned data from row to row for the other data lengths - ie, the data widths 100, 50 and 20 are not divisible by 8. To show the effect of this we looked at the following square matrices:

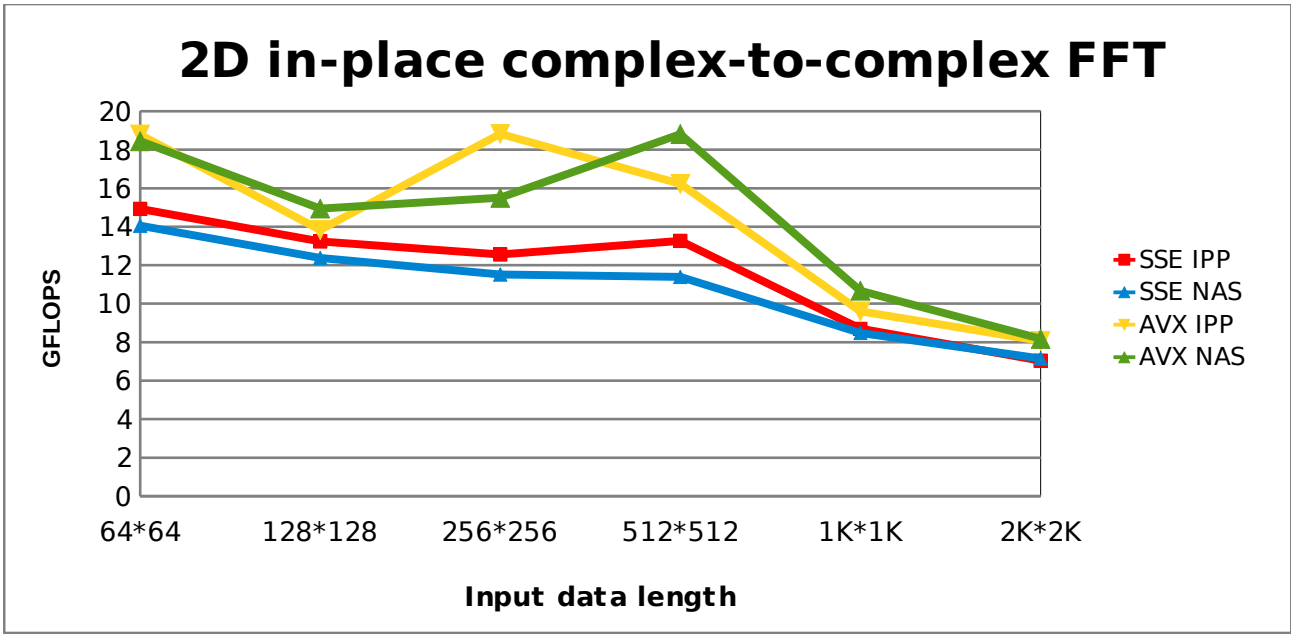
Table 4: vsip_ccfft2dip_f timings.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
SSE IPP	16.5	86.7	418	1,779	12,062	65,622
SSE NAS	17.5	92.7	455	2,071	12,364	64,485
AVX IPP	13.1	82.9	279	1,454	10,927	57,257
AVX NAS	13.3	76.8	338	1,254	9,825	56,464

Here we are studying square complex matrices from 64 * 64 complex cells all the way up to matrices of 2024 * 2048 complex cells. The timing results show a significant speed up across all square matrices sizes. The following graph highlights this performance gain.

Figure 3: complex to complex 2D in-place FFT performance.

MFLOPS = $5NLog_2(N)*M + 5MLog_2(M)*N$ / (time for one FFT in microseconds)



The graphs above show that AVX IPP has a performance advantage over NAS for a data size of 256 by 256. Other data sizes show that AVX NAS has an advantage. The results also highlight that AVX has a substantial performance improvement across all these data sizes for both IPP and NAS. At 512*512 the performance gain is 79% for NAS. These tests also suggest to us that the 2d FFT algorithms can be improved for unaligned data and non-square data sizes.

The performance improvement is much better for these data sizes. This is due to the following reasons:

- Where the data sizes are smaller the algorithm is less memory dependent. As the data sizes get bigger they do not fit into cache and cache misses become more significant.

- The 2d FFT algorithm is more efficient at present with square matrices as opposed to non-square matrices.

- All the data from row-to-row and column-to-column is aligned (the length and width is dividable by 8).

3.3. *Multiple in-place complex-to-complex FFT.*

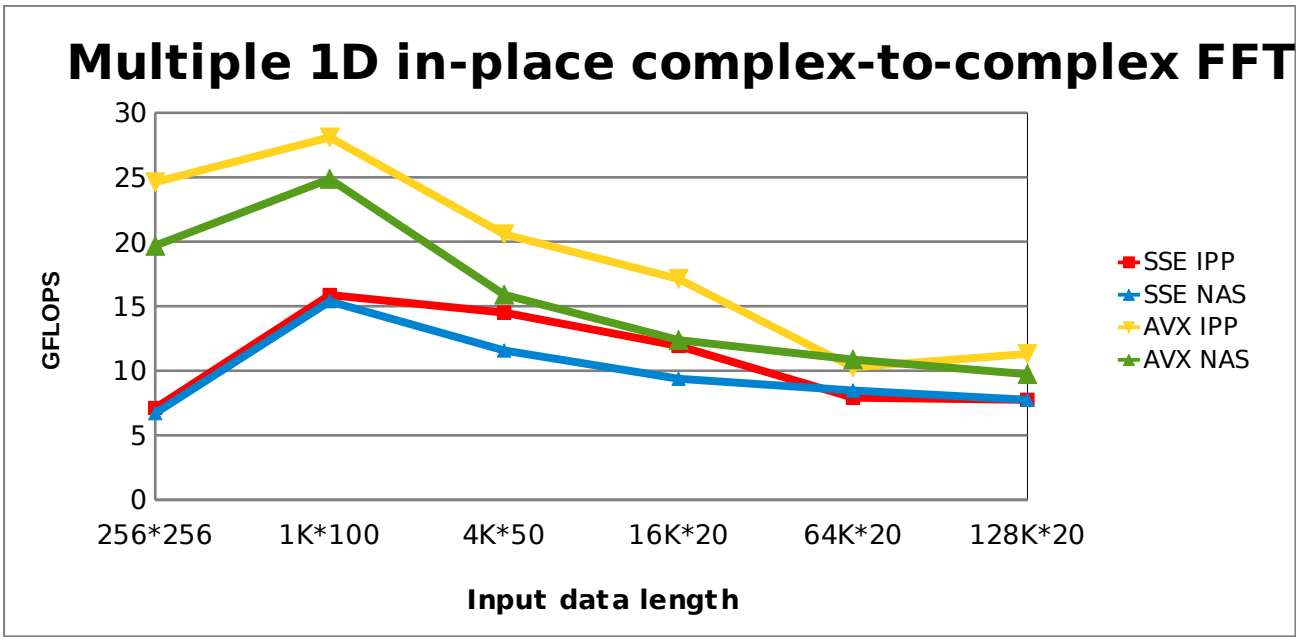
The multiple in-place complex-to-complex operation is carried out by the library call vsip_ccfftmip_f. This function gets called after VSIPL data matrix views and FFT plans have been created. The following table and graph show the timing results of running the set benchmark parameters. The results in the table are in microseconds and the results in the graph are in GFlops.

Table 5: vsip_ccfftmip_f timings.

Input Data Length	256*256	1K*100	4K*50	16K*20	64K*20	128K*20
SSE IPP	166	315	827	1,880	12,942	28,101
SSE NAS	189	325	1,037	2,390	12,105	28,079
AVX IPP	104	178	583	1,309	9,979	19,272
AVX NAS	130	201	755	1,814	9,416	22,361

Figure 4: complex to complex multiple in-place FFT performance.

MFLOPS = $5NLog_2(N)*M$ / (time for one FFT in microseconds)



The above graph shows that the performance improvement for 1K*100 for the NAS code is 77% and for IPP based code it is 87%. For larger data lengths the improvement for AVX code is still very significant Over the data lengths used, the average performance improvement was 54% for NAS code and 65% for IPP.

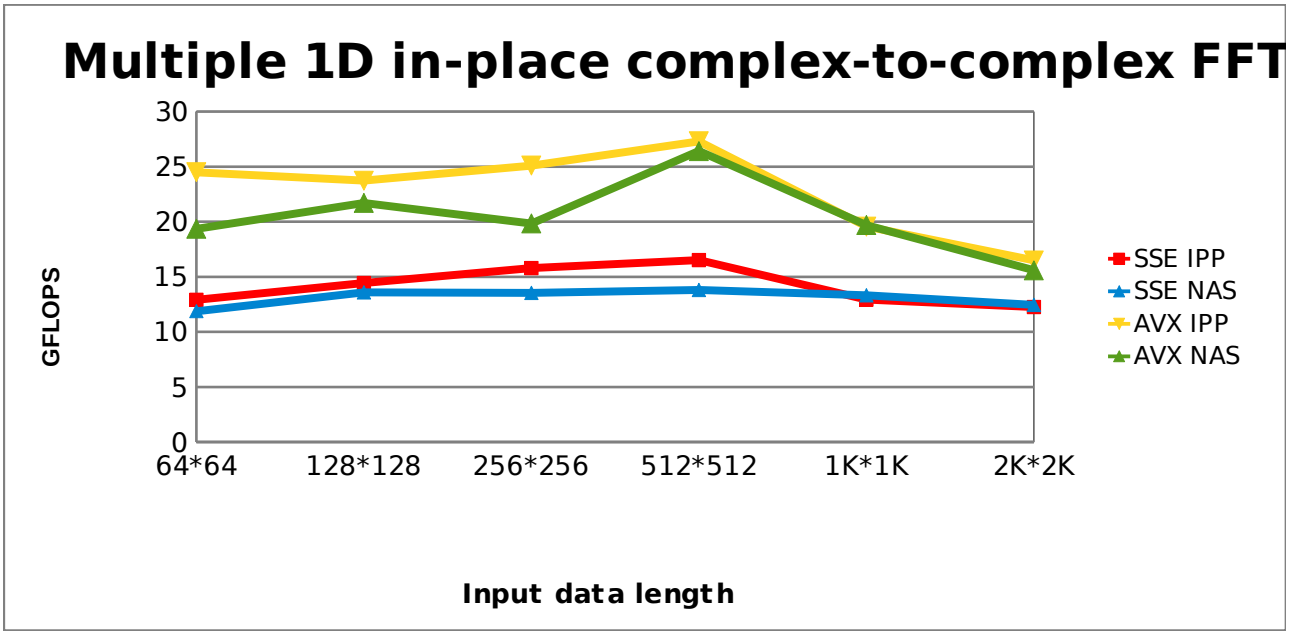
The following table shows timings for multiple complex to complex FFTs using square matrices. Results in the table are in microseconds and the results in the graph are in GFlops.

Table 6: vsip_ccfftmip_f timings.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
SSE IPP	9.3	38.8	162	697	3,957	18,382
SSE NAS	10.1	41.3	189	834	3,843	18,100
AVX IPP	4.9	23.6	102	422	2,617	13,657
AVX NAS	6.2	25.8	129	436	2,599	14,454

Figure 5: complex to complex multiple in-place FFT performance.

MFLOPS = $5NLog_2(N)*M + 5MLog_2(M)*N$ / (time for one FFT in microseconds)



The above graph shows that the performance improvement for 512*512 for the NAS code is 95%. For 64*64 the improvement for IPP is 96%. Over the data lengths used, the average performance improvement was 69% for NAS code and 74% for IPP.

3.4. Complex matrix transpose.

A complex matrix transpose operation is performed in VSIPL by calling the function vsip_cmtrans_f. This function is given a VSIPL view representing an input data matrix and the function transposes the views data into an output view. The operation is therefore performed out-of-place. The following table shows the timings obtained for SSE/AVX and IPP/NAS. The timings are given in microseconds.

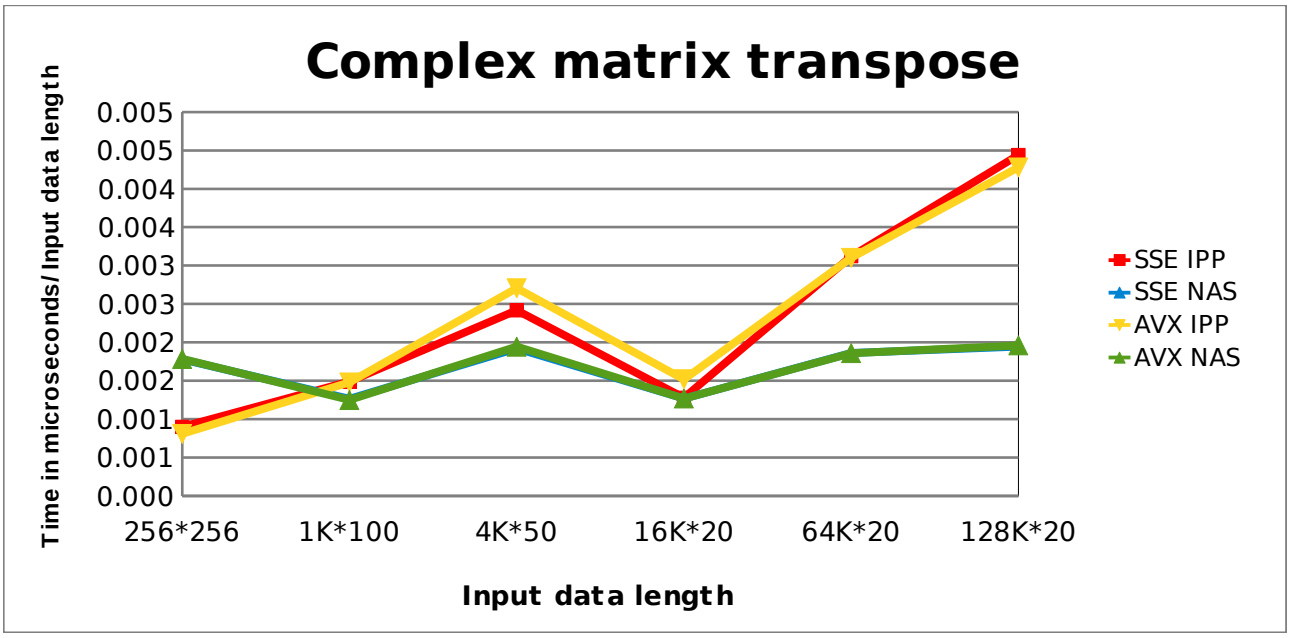
Table 7: vsip_cmtrans_f timings.

Input Data Length	256*256	1K*100	4K*50	16K*20	64K*20	128K*20
SSE IPP	59	153	495	419	4,091	11,642
SSE NAS	116	129	394	415	2,439	5,100
AVX IPP	53	151	554	498	4,063	11,217
AVX NAS	117	128	398	415	2,436	5,137

These timings show that IPP is better than NAS at 256 by 256. However, NAS is substantially better than IPP for the other data lengths. This is because NAS has an optimized algorithm for misaligned data within the transpose operation. Neither NAS or IPP show a significant increase in performance between SSE and AVX for non-squared matrices. This is largely down to the fact that the operation is mainly memory dependent and is not computationally expensive. The IPP library showed an AVX performance improvement of 27% for the square matrix 256 by 256. The graph below shows the performance in time divided by data size. So the fastest algorithm is now the lowest line in the graph. (Note, the blue line is almost hidden by the green line).

Figure 6: complex matrix transpose performance.

Graph shows time in microseconds / data size



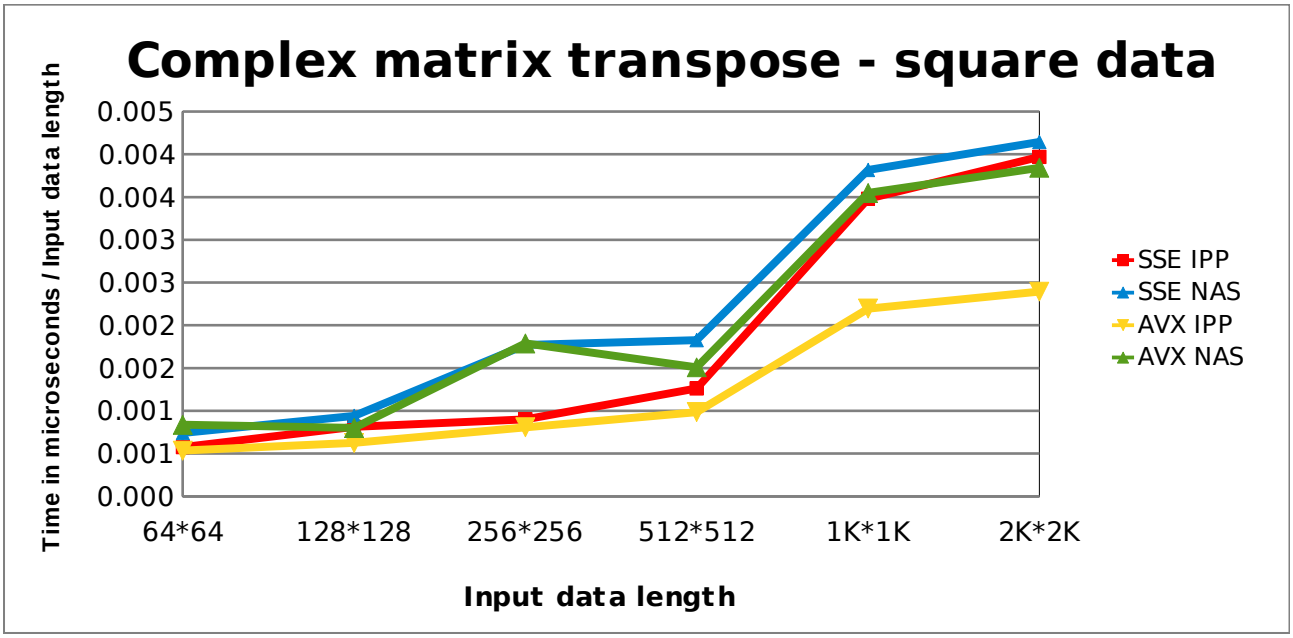
As with the 2d FFTs the efficiency of the algorithms are not as good as it would be with square matrices due to the data sizes being misaligned from row-to-row. The results also show a big difference in performance between NAS and IPP for both SSE and AVX with NAS being the most efficient choice for non-square matrices. The table below shows a series of square matrix times from a complex matrix of 64 by 64 cells up to 2048 by 2048 cells.

Table 8: vsip_cmtrans_f square matrix timings.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
SSE IPP	2.36	13.34	59.0	331	3,649	16,648
SSE NAS	3.04	15.42	116.0	479	4,002	17,383
AVX IPP	2.19	10.24	53.0	259	2,303	10,042
AVX NAS	3.43	13.10	117.2	395	3,720	16,118

Figure 7: complex matrix transpose performance.

Graph shows time in microseconds / data size



The above graphs show the execution time divided by the data size with fastest algorithms lower in the graphs. The slowest time is for SSE NAS shown in blue. However on square matrices the IPP AVX library is well optimized and ends up with the fastest algorithm shown in yellow. The IPP library has a good AVX performance across most of the data sizes considering the transpose operation is not computationally expensive. At 2K by 2K the IPP library has a AVX performance of 79%. The NAS library is well optimized for non-square matrices but as the above graph shows it needs further work to optimize it for AVX square transpose operations.

3.5. Complex Vector Multiply.

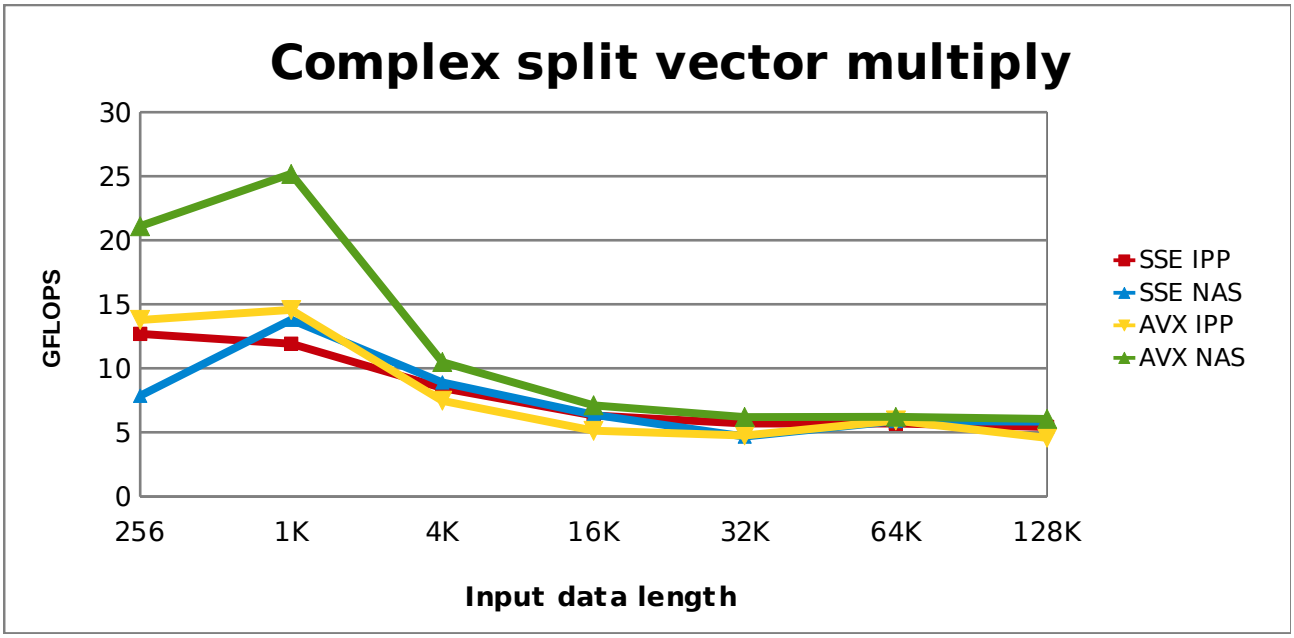
Complex vector multiply is performed in the VSIPL library by a call to the function vsip_cvmul_f. In this test one complex split vector is multiplied with a second and the result is placed in a third output vector. The following table and graph show the optimized SSE and AVX timings. The table shows the results in microseconds and the graph has them in GFlops.

Table 9: vsip_cvmul_f timings.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP	0.12	0.52	2.90	15.53	34.54	68.9	144.8
SSE NAS	0.20	0.45	2.76	15.39	41.98	66.9	134.6
AVX IPP	0.11	0.42	3.29	19.18	41.30	66.2	172.1
AVX NAS	0.07	0.24	2.34	13.82	31.69	63.2	130.0

Figure 8: Split complex vector multiply performance.

MFLOPS = 6 * N / (time for one vector multiply in microseconds)



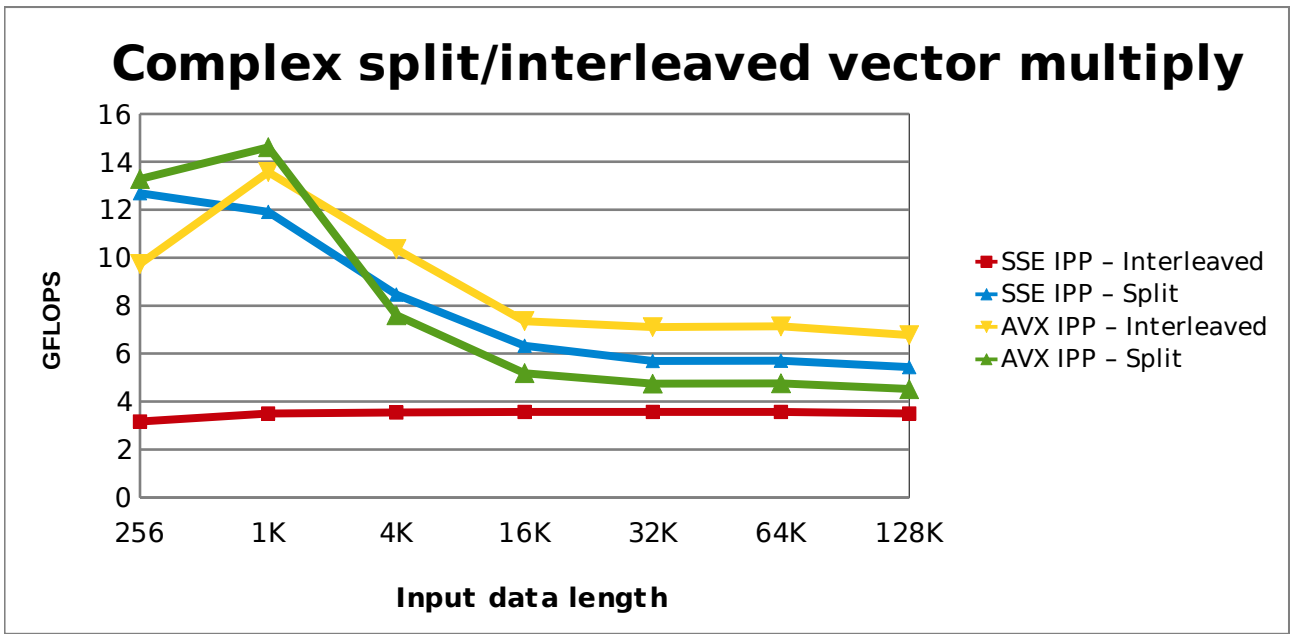
The above graph shows the NAS SSE time in blue and the NAS AVX time in green. This shows a good AVX performance up to a data length of 4K with the performance then narrowing. After 4K the algorithm becomes more memory dependent and the difference between AVX and SSE narrows as expected for this operation. At a data length of 1K the AVS performance is 93%. The IPP library has a good SSE performance but is not well optimized in the above graph for AVX. The above tests perform a split complex multiply operation. To make sure we were being fair to IPP we timed both split and interleaved complex multiply operations using the IPP library. The results are shown below.

Table 10: Complex split / interleaved vector multiply using IPP.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP – Interleaved	0.48	1.75	6.92	27.56	55.11	110.04	224.42
SSE IPP – Split	0.12	0.52	2.9	15.53	34.54	68.94	144.77
AVX IPP – Interleaved	0.16	0.45	2.37	13.37	27.67	55.1	116.11
AVX IPP – Split	0.12	0.42	3.23	18.95	41.39	82.53	173.89

Figure 9: Split/Interleaved complex vector multiply performance using IPP.

MFLOPS = 6 * N / (time for one vector multiply in microseconds)



The graph shows the worst time obtained is for SSE interleaved shown in red. The AVX interleaved time is shown in yellow and highlights that the algorithm is AVX optimized. The performance for AVX split and AVX interleaved peaks for a data length of 1K at 14.8 GFlops. However, this is significantly less than the NAS performance of 25 GFlops shown in the previous graph.

3.6. Vector Sine.

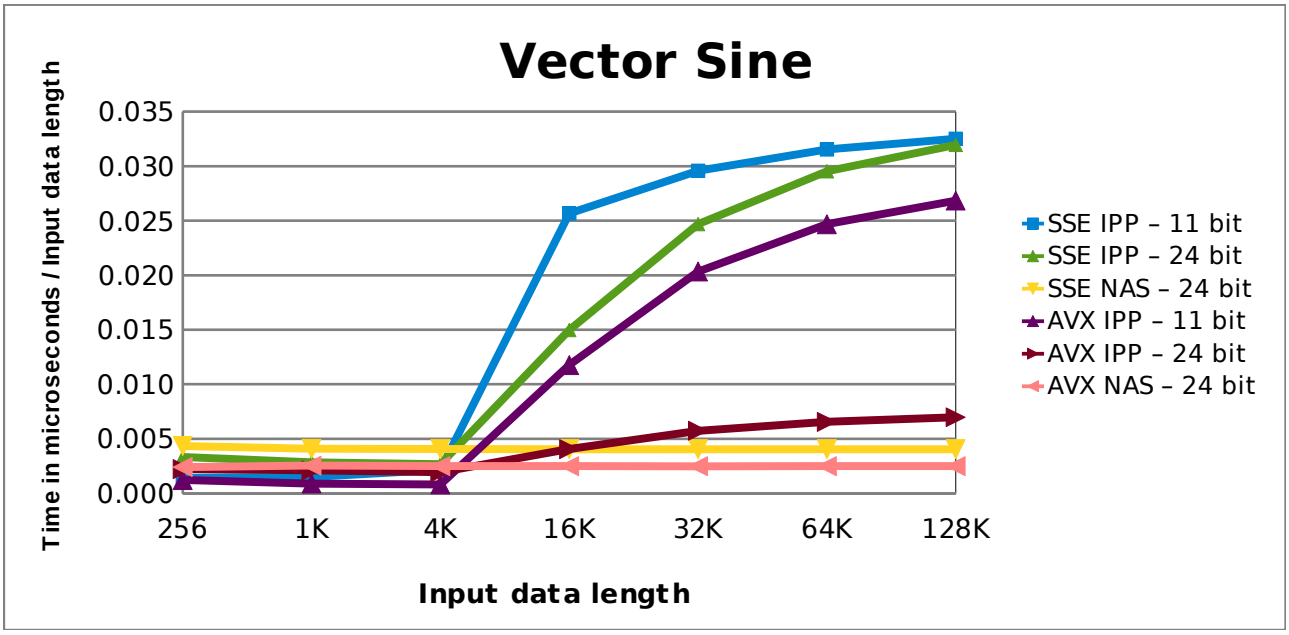
The vector sine operation is performed in the VSIPL library by a call to the function vsip_vsin_f. In this test the sine is taken of each element within a floating point input vector and the result is placed into an output vector. The following table and graph show the optimized SSE and AVX. The table has timings in microseconds and the graph displays the input data size divided by the time in microseconds. The IPP library has different floating point accuracies for this operation. The NAS call is always a full accuracy of 24 bits.

Table 11: vsip_vsin_f timings.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP – 11 bit	0.36	1.53	8.83	420	969	2,066	4,262
SSE IPP – 24 bit	0.86	2.91	10.96	246	809	1,936	4,188
SSE NAS – 24 bit	1.11	4.18	16.63	66	133	266	531
AVX IPP – 11 bit	0.31	0.91	3.34	193	667	1,618	3,518
AVX IPP – 24 bit	0.57	2.15	8.00	67	188	429	915
AVX NAS – 24 bit	0.62	2.61	10.20	41	82	163	327

Figure 10: vector sine performance.

Graph shows time in microseconds / data length



The yellow graph shows the NAS SSE time and the pink graph below shows the NAS AVX time. This has a good AVX performance across all lengths. At a data length of 256 the AVX performance of the NAS library is 88%. The algorithm for both NAS and IPP carries out a full range reduction to make sure the input values are in the correct quadrant. This means the algorithm is more computationally expensive than say the vector multiply operation. This in turn means the AVX performance is more constant across all data lengths. The best performance obtained above is for the AVX NAS algorithm and this applies across all lengths. The IPP performance is good until we reach 4K and then the timings increase substantially.

The AVX performance here is impressive because of the number of integer operations being carried out. The algorithm uses integer operations in order to carry out the range reduction to define the quadrant that the input values lie in. Unfortunately, many of the integer operations are carried out with 4way SIMD instructions because of the lack of support of 8way SIMD integer instructions within the AVX instruction set. Despite the lack of 8way SIMD integer instructions the results presented above look very impressive.

3.7. Vector Cosine.

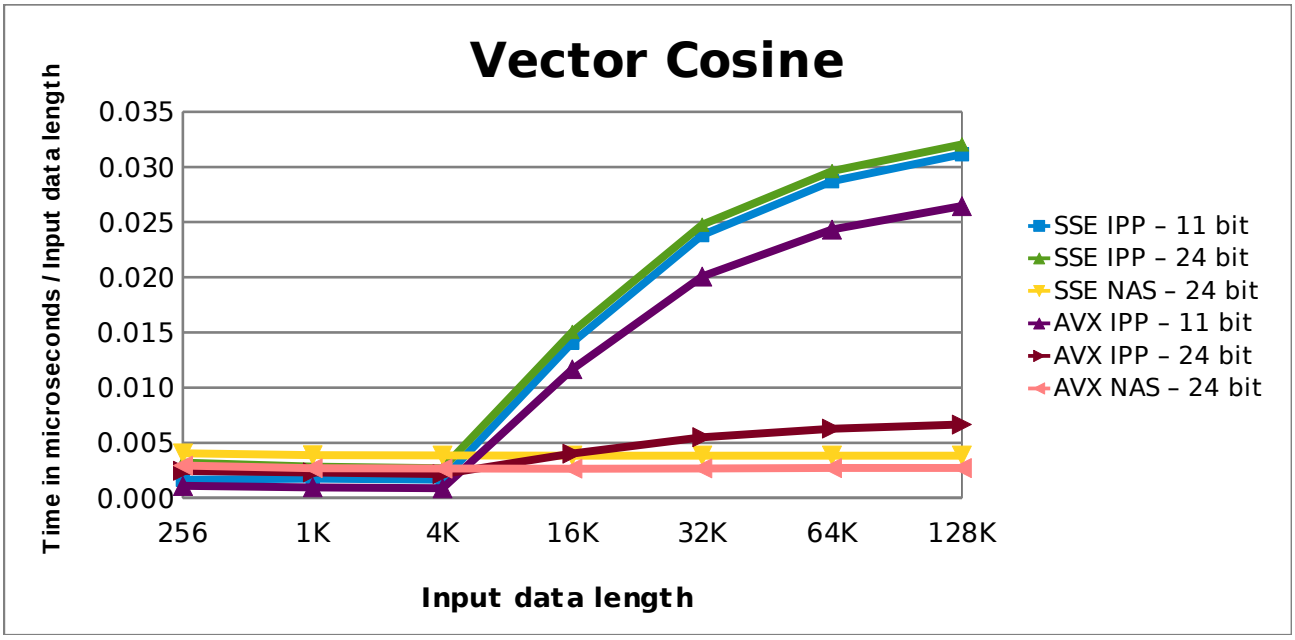
The vector cosine operation is performed in the VSIPL library by a call to the function vsip_vcos_f. In this test the cosine is taken of each element in a floating point input vector and the result is placed into an output vector. The following table and graph show the optimized SSE and AVX timings. The table is in microseconds and the graph displays the input data size divided by the time in microseconds. In data below we have given results for IPP operating with a low accuracy of 11 bits and both IPP/NAS calling high accuracy algorithms of 24 bits.

Table 12: vsip_vcos_f timings.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP – 11 bit	0.43	1.80	6.80	231	781	1,881	4,083
SSE IPP – 24 bit	0.82	2.90	11.02	247	811	1,942	4,201
SSE NAS – 24 bit	1.03	3.96	15.74	63	126	251	502
AVX IPP – 11 bit	0.28	0.98	3.57	191	658	1,594	3,468
AVX IPP – 24 bit	0.63	2.36	8.87	66	181	411	871
AVX NAS – 24 bit	0.75	2.75	10.87	43	88	177	355

Figure 11: vector cosine performance.

Graph shows time in microseconds / data length



The vector cosine operation has a similar graph to the vector sin operation. The AVX performance of the NAS cosine operation is fairly constant across all data lengths. The AVX performance at a data length of 1K is 61%. As was the case with the sine test these results look very good given the lack of support of 8way SIMD operations within the AVX instruction set.

3.8. Vector Square Root.

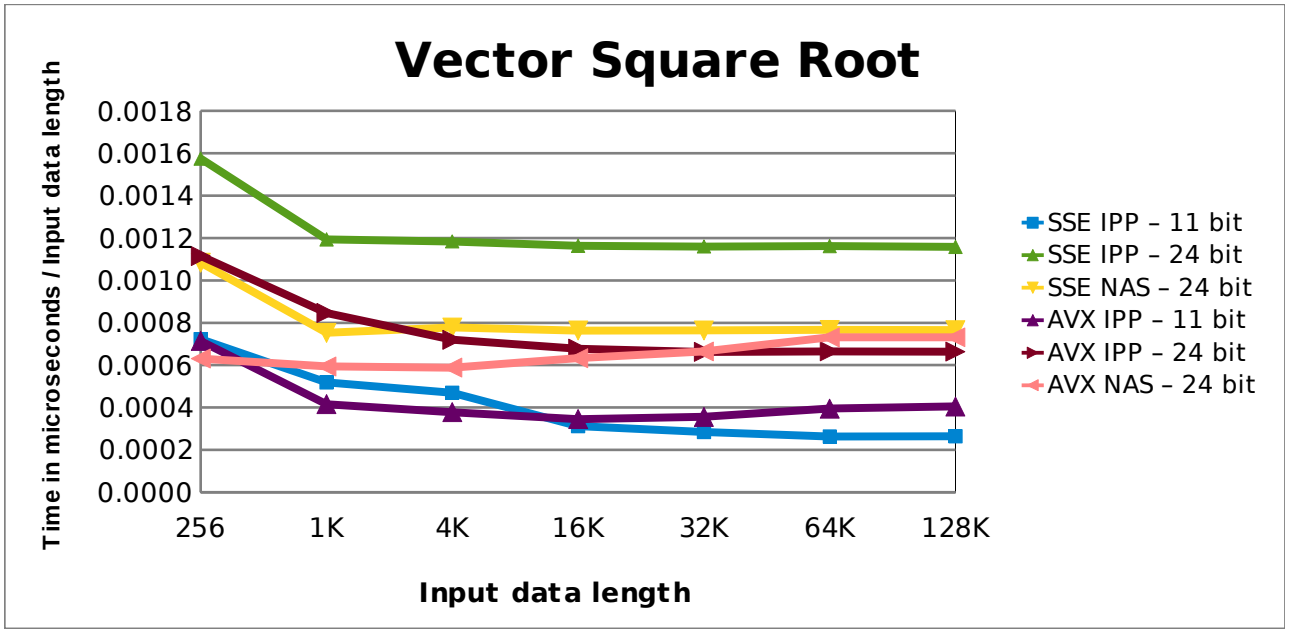
The vector square root operation is performed in the VSIPL library by a call to the function vsip_vsqrt_f. In this test the square root is taken of each element in a floating point input vector and the result is placed into an output vector. The following table and graph show the optimized SSE and AVX timings. The table is in microseconds and the graph displays the input data size divided by the time in microseconds. The NAS algorithm has a full floating point accuracy of 24 bits. The IPP library has different accuracies for the square root function. Accuracies 11 and 24 bits have been given in the table below.

Table 13: vsip_vsqrt_f timings.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP – 11 bit	0.18	0.53	1.92	5.11	9.32	17.24	34.63
SSE IPP – 24 bit	0.40	1.22	4.85	19.06	38.02	76.13	151.84
SSE NAS – 24 bit	0.28	0.77	3.19	12.50	25.04	50.21	100.28
AVX IPP – 11 bit	0.18	0.43	1.55	5.66	11.66	25.89	53.05
AVX IPP – 24 bit	0.29	0.87	2.95	11.11	21.70	43.58	87.01
AVX NAS – 24 bit	0.16	0.61	2.41	10.37	21.78	47.91	95.84

Figure 12: vector square root performance.

Graph shows time in microseconds / data length



The two bottom graphs in blue and purple are for IPP 11 bit accuracy. These produce the fastest answer but are not at accurate as the other graphs. The AVX performance also turns negative about 8K.

The yellow and pink graphs represent the NAS square root operations with the yellow graph being the SSE performance and the pink graph representing the AVX performance. Here we get a good but narrowing AVX performance across all data lengths. The performance narrows because the algorithm becomes more memory dependent as the vector size increases. The square root operation is not as computationally expensive as the sin and cosine operations so memory effects are more significant. At a vector length of 256 the AVX performance is 86%.

The graphs shown in brown and green at the top represent the IPP with 24 bit accuracy. This is the same accuracy as the NAS routines. The AVX performance is good across all lengths.

3.9. Vector Scatter.

The vector scatter operation usually uses a small input vector and a large output vector. The smaller input vector also comes with an input vector of indexes stating how its elements should be scattered into the larger vector. The VSIPL library performs the vector scatter operation with a call to vsip_vscatter_f. In the tests below the larger vector is twice the size of the smaller vector and the elements are scattered into every other element of the larger vector. The data lengths below give the input vector size. The code in the SSE version was serial code because of the simple nature of the operation. The table below shows the serial SSE time and the serial AVX time. The AVX compiler optimization produced a significant improvement in performance by just compiling the serial code. At this point NAS optimized the algorithm to take advantage of AVX 8way SIMD floating point load instructions given that the data in the input vectors are aligned in these tests. The last row in the table below shows this AVX optimized time. The timings in the table are given in microseconds.

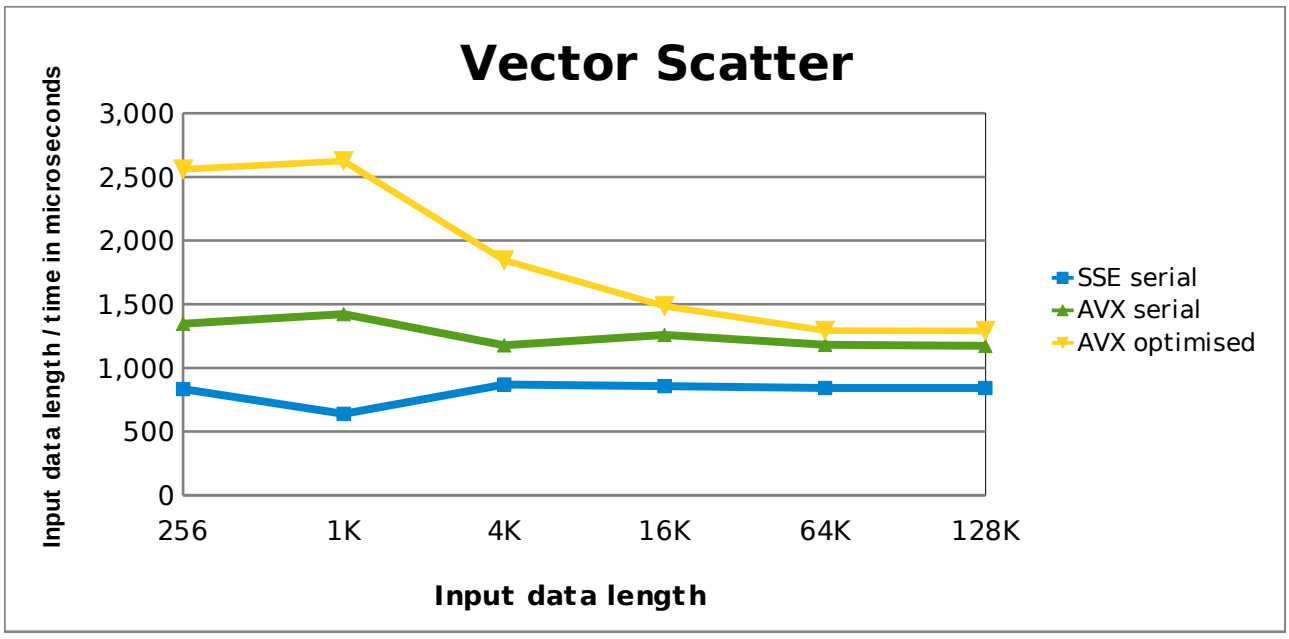
Input Data Length	256	1K	4K	16K	64K	128K
SSE serial	0.31	1.60	4.71	19.11	77.79	155.51
AVX serial	0.19	0.72	3.48	13.00	55.50	111.75
AVX Optimized	0.1	0.39	2.22	11.03	50.69	101.52

Table 14: vsip_vscatter_f timings.

The above timings show that the optimisation by the compiler produced a speed up of 1.6 at 256 and 2.2 at 1K. This reduced down to a rate of 1.39 at 128K. The AVX hand coded optimisations then produced another speed up 1.85 in performance at 1K. Given that the best ideal world AVX performance would be to half the execution time, the AVX performance is 92% of this at 1K. Going from 0.72 microseconds to 0.39 microseconds. The following graph shows the input data size divided by the time in microseconds to highlight the performance of the algorithm on the AVX machine.

Figure 13: vector scatter performance.

Graph shows data length / time in microseconds



3.10. Vector Gather.

The vector gather operation usually gathers elements from a large input vector into a smaller output vector. The smaller output vector comes with a vector of indexes stating the position of the elements in the larger vector. The output vector and vector of indexes are the same size and are given in the table below. The table shows the SSE serial time, the AVX serial time and the AVX optimised time.

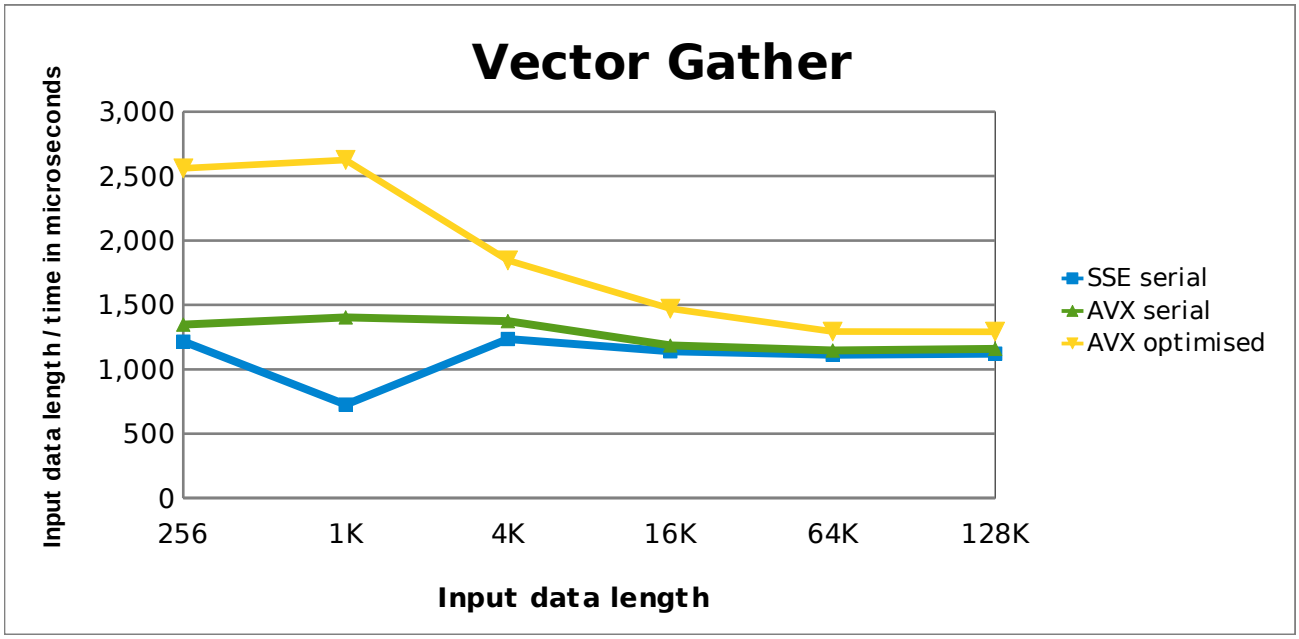
Table 15: vsip_vgather_f timings.

Input Data Length	256	1K	4K	16K	64K	128K
SSE serial	0.21	1.41	3.32	14.39	58.88	116.96
AVX serial	0.19	0.73	2.98	13.84	57.21	113.13
AVX optimized	0.1	0.39	2.22	11.14	50.7	101.49

The compiler optimisations are slightly less than the vector scatter test but are very still significant. The largest speed up from serial SSE code to optimised AVX code is at 1K and is 1.93. The AVX performance is just as good as the vector scatter operation with a figure of 93% at 1K. The graph below shows the input data size divided by the time in microseconds.

Figure 14: vector gather performance.

Graph shows data length / time in microseconds



4. The Arrandale SSE and AVX Performance.

In Section 3 of this report we compared the AVX performance of a number of vsipl operations to the SSE performance. In order to carry out the study two lots of test programs were produced. One set was compiled for SSE and the other set for AVX. In section 3 both the SSE and AVX programs were then executed and timed on the same AVX test platform. This was possible because the AVX platform is fully backward compatible with SSE. The timings produced then showed the performance advantage of using the AVX instruction set on the AVX platform. This performance advantage was due to:

- The AVX platform having 256 bit registers allow 8 way SIMD operations.
- The AVX platform having a larger richer instruction set including:
  - Broadcast instructions;
  - Data permute instructions;
  - 256 and 128 bit SIMD instructions;

However, the AVX platform itself has a number of advantages over previous SSE platforms such as a higher bandwidth for loads and stores that the previous section did not highlight. The AVX platform has an improved memory controller and two 128-bit load ports. This means the memory speed is faster than a typical SSE platform. In defense/DSP applications such as SAR, Sonar, MTI, Image processing and medical scanning we are quite often memory bound due to the large amounts of data being processed. This means the AVX platform is particularly well suited to the task due to the improved CPU and memory performance.

In this section we compare the AVX test programs running on the AVX platform to the SSE test programs running on one of the latest SSE platforms. To do this we have used the Intel SSE4 Arrandale i5 platform. Our Arrandale platform runs at a maximum operating frequency of 2.4 GHz. For the purposes of this study we have set the operating frequency to 2.0 GHz. This is because our AVX test platform is 2.0GHz and therefore we are comparing like with like.

The following subsections describe the performance of the same VSIPL operations that were given in section 3. However, this time we are comparing the Arrandale performance to the AVX test platform performance. The AVX timings given in this section are the same AVX timings given in section 3. However, the SSE timings have been taken using the Arrandale platform.

4.1. 1D in-place complex-to-complex FFT.

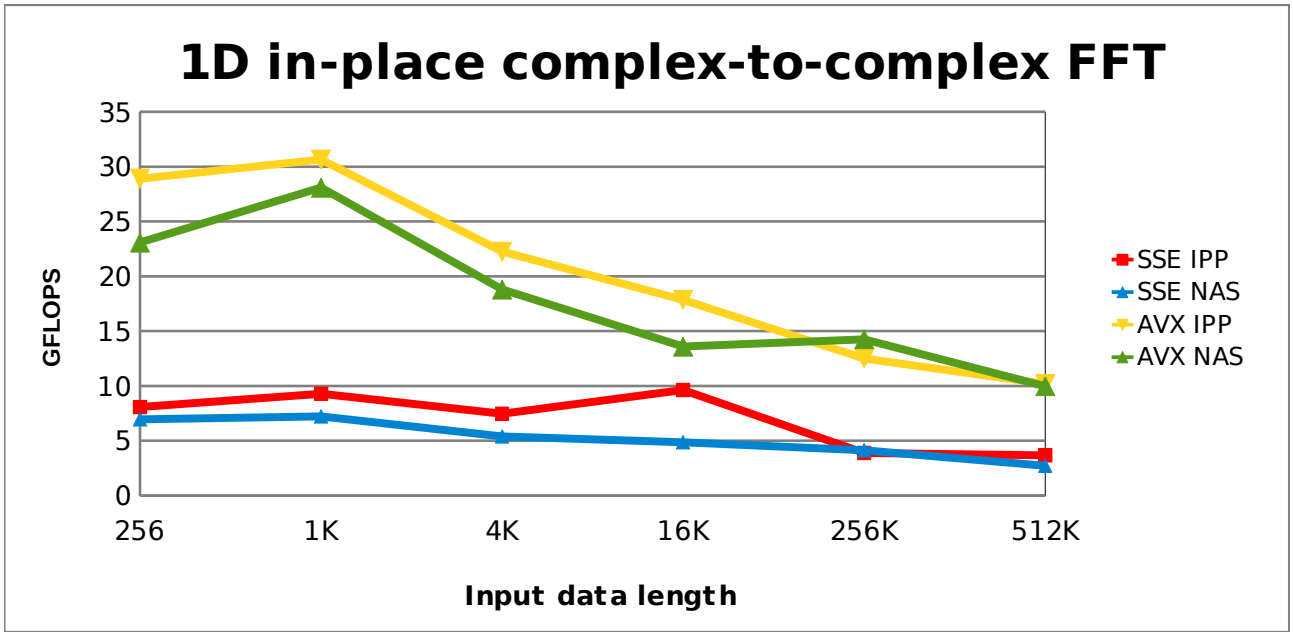
The following table and graph show the obtained timings when the SSE and AVX timing programs were executed on the Arrandale and AVX platforms. We give here both NAS based timings and timings based on Intel's IPP library. The results in the table are in microseconds and the results in the graph are presented in GFlops.

Table 16: vsip_ccfftip_f timings.

Input Data Length	256	1K	4K	16K	256K	512K
SSE IPP	1.24	5.38	32.14	116.4	5,950	13,290
SSE NAS	1.44	6.94	44.59	230.8	5,601	18,027
AVX IPP	0.35	1.67	11.05	64.3	1,888	4,890
AVX NAS	0.44	1.82	13.08	84.5	1,656	4,999

Figure 15: complex to complex 1D in-place FFT performance.

MFLOPS = 5 N Log_2(N) / (time for one FFT in microseconds)



The red and blue graphs show the performance of the 1d FFTs on the Arrandale platform. The yellow and green graphs show the AVX performance with the yellow graph highlighting the IPP library. Both the IPP and NAS based timings show huge performance improvements on AVX. When all test programs were executed on the same AVX platform no speed ups were greater than twice as quick. We see much greater speed ups in the above graph when comparing the Arrandale platform to the AVX platform. The table below shows the rate of speed up for both NAS and IPP with the NAS 1K FFT being 3.82 times quicker on the AVX platform.

Table 17: vsip_ccfftip_f speed ups.

Input Data Length	256	1K	4K	16K	256K	512K
IPP Speedup Rate.	3.50	3.22	2.90	1.81	3.15	2.72
NAS Speedup Rate.	3.27	3.82	3.41	2.73	3.54	3.70

4.2. 2D in-place complex-to-complex FFT.

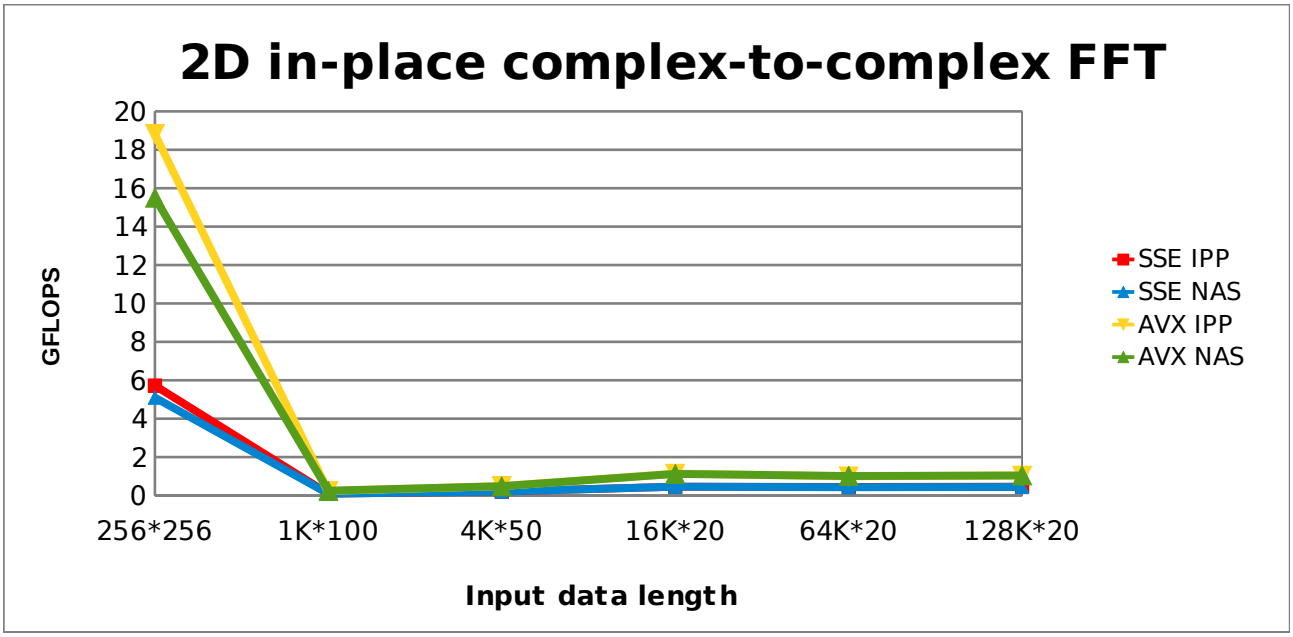
The following table and graph show the timing results of running the set benchmark parameters for the 2d in-place complex-to-complex FFT. The results in the table are in microseconds and the results in the graph are in GFlops.

Table 18: vsip_ccfft2dip_f timings.

Input Data Length	256*256	1K*100	4K*50	16K*20	64K*20	128K*20
SSE IPP	853	81,891	84,964	63,805	292,110	586,851
SSE NAS	956	83,331	85,307	64,405	303,901	614,801
AVX IPP	279	35,956	36,928	26,323	133,607	269,949
AVX NAS	338	34,655	37,018	26,824	130,293	267,780

Figure 16: complex to complex 2D in-place FFT performance.

MFLOPS = $5NLog_2(N)*M + 5MLog_2(M)*N$ / (time for one FFT in microseconds)



You can see from the graph the matrix 256 by 256 is significantly faster for AVX. The other matrices are not as efficient due to the matrix data sizes being non-square. However, the table below shows the speed up even for these data sizes is about 2.3 times quicker for AVX. The square matrix 256 by 256 is 3.06 times quicker for IPP on AVX due to the 8way SIMD implementation.

Table 19: vsip_ccfft2dip_f speed ups.

Input Data Length	256*256	1K*100	4K*50	16K*20	64K*20	128K*20
IPP Speedup Rate.	3.06	2.30	2.30	2.42	2.19	2.17
NAS Speedup Rate.	2.83	2.40	2.30	2.40	2.33	2.30

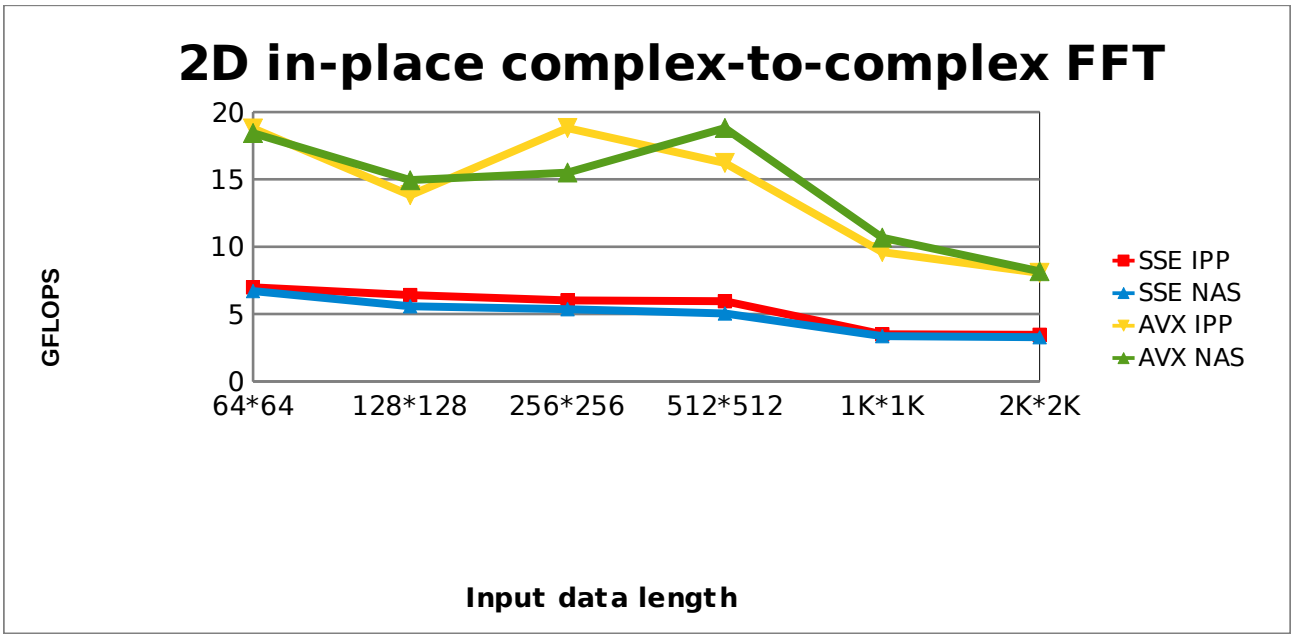
The table below shows the 2d FFT times over a series of square matrices. These data sizes are all aligned by 8 from row to row. This means we can take advantage of the full 8-way SIMD instructions without having to handle misaligned data.

Table 20: vsip_ccfft2dip_f square matrix timings.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
SSE IPP	34.4	174.6	853	3,872	29,183	130,519
SSE NAS	35.8	200.7	956	4,562	30,507	137,871
AVX IPP	13.1	82.9	279	1,454	10,927	57,257
AVX NAS	13.3	76.8	338	1,254	9,825	56,464

Figure 17: complex to complex square 2D in-place FFT performance.

MFLOPS = $5NLog_2(N)*M + 5MLog_2(M)*N$ / (time for one FFT in microseconds)



The graphs above show that AVX IPP has a performance advantage over NAS for a data size of 256 by 256. Other data sizes show that AVX NAS has an advantage. The results also highlight that AVX has a substantial performance improvement across all these data sizes for both IPP and NAS. The table below shows that at a matrix size of 512 by 512 the NAS algorithm is 3.64 times quicker on the AVX platform.

Table 21: vsip_ccfft2dip_f square matrix speed ups.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
IPP Speedup Rate.	2.63	2.11	3.05	2.66	2.69	2.28
NAS Speedup Rate.	2.69	2.61	2.83	3.64	3.11	2.44

4.3. Multiple in-place complex-to-complex FFT.

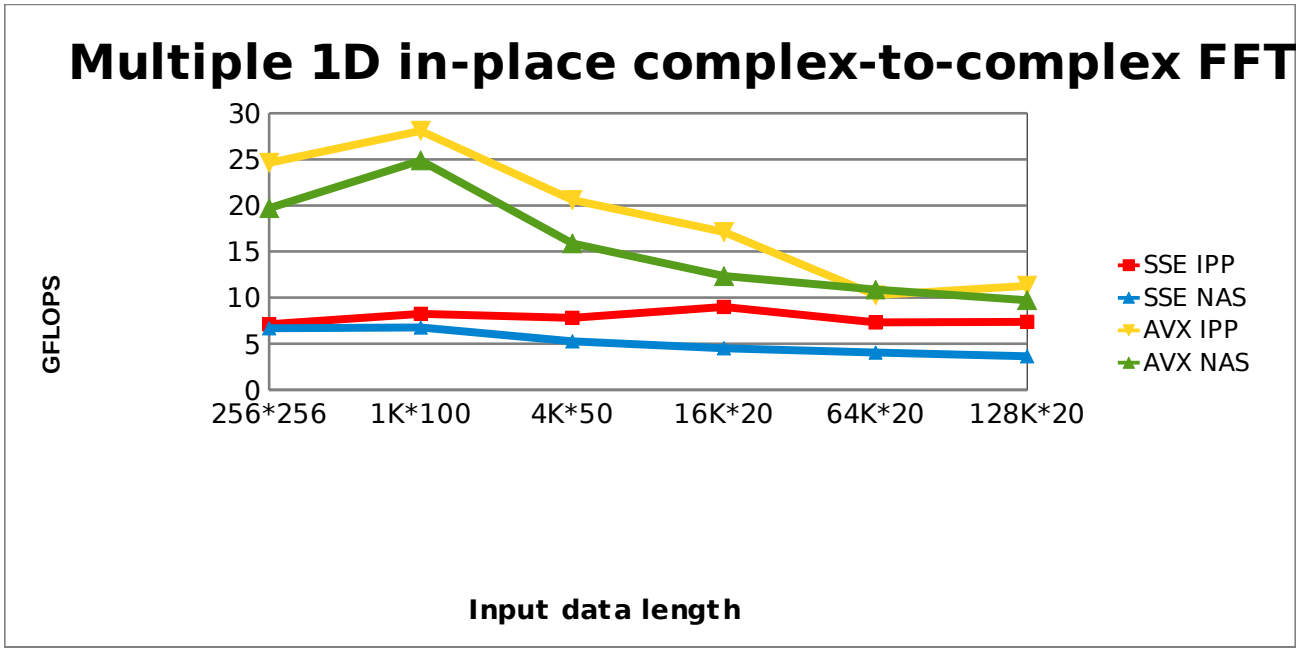
The following table and graph show the timing results of running the set benchmark parameters for the multiple in-place complex-to-complex FFT. In these tests the rows of each matrix are being FFTed. The results in the table are in microseconds and the results in the graph are in GFlops.

Table 22: vsip_ccfftmip_f timings.

Input Data Length	256*256	1K*100	4K*50	16K*20	64K*20	128K*20
SSE IPP	360	606	1,534	2,488	13,979	29,486
SSE NAS	382	738	2,271	4,945	25,343	59,462
AVX IPP	104	178	583	1,309	9,979	19,272
AVX NAS	130	201	755	1,814	9,416	22,361

Figure 18: complex to complex multiple in-place FFT performance.

MFLOPS = $5NLog_2(N)*M$ / (time for one FFT in microseconds)



The blue and red graphs above show the SSE performance on Arrandale. The green and yellow graphs show the AVX performance and the large performance advantage of AVX. The table below shows the speed ups over all data lengths with the NAS 1K by 100 working out as 3.67 times quicker for AVX.

Table 23: vsip_ccfftmip_f speed ups.

Input Data Length	256*256	1K*100	4K*50	16K*20	64K*20	128K*20
IPP Speedup Rate.	3.46	3.40	2.60	1.90	1.40	1.53
NAS Speedup Rate.	2.94	3.67	3.01	2.73	2.69	2.66

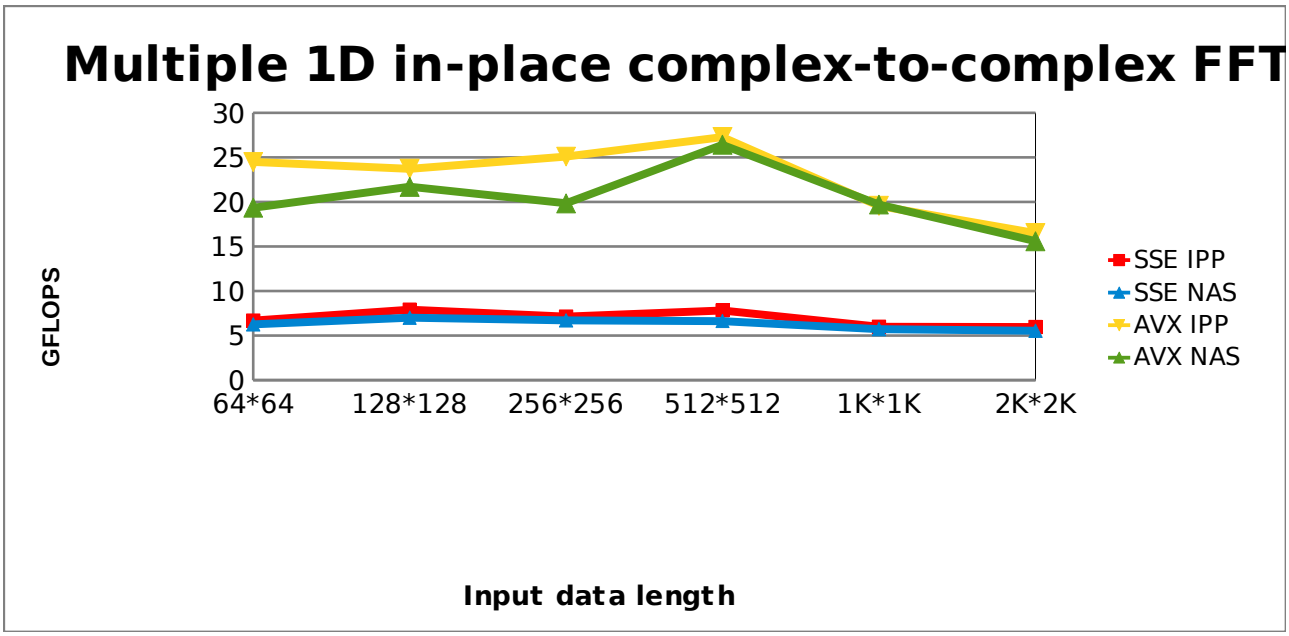
The following table shows timings for multiple complex to complex FFTs using square matrices. Results in the table are in microseconds and the results in the graph are in GFlops.

Table 24: vsip_ccfftmip_f square matrix timings.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
SSE IPP	18.0	70.9	360	1,474	8,554	37,847
SSE NAS	19.2	80.2	382	1,741	8,995	40,775
AVX IPP	4.9	23.6	102	422	2,617	13,657
AVX NAS	6.2	25.8	129	436	2,599	14,454

Figure 19: complex to complex multiple in-place FFT performance.

MFLOPS = $5NLog_2(N)*M$ / (time for one FFT in microseconds)



The above graph shows a larger performance improvement for AVX than the non-square results contained in Figure 18. This is because all the data in the square matrices shown above are aligned. ie, the data lengths are a multiple of eight. The following table shows the speed ups for each data size with the NAS 512 by 512 working out as 3.99 times quicker than SSE:

Table 25: vsip_ccfftmip_f square matrix speed ups.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
IPP Speedup Rate.	3.67	3.00	3.53	3.49	3.27	2.77
NAS Speedup Rate.	3.10	3.11	2.96	3.99	3.46	2.82

4.4. Complex matrix transpose.

The following table shows the timings obtained for the complex matrix transpose operation. The timings are given in microseconds.

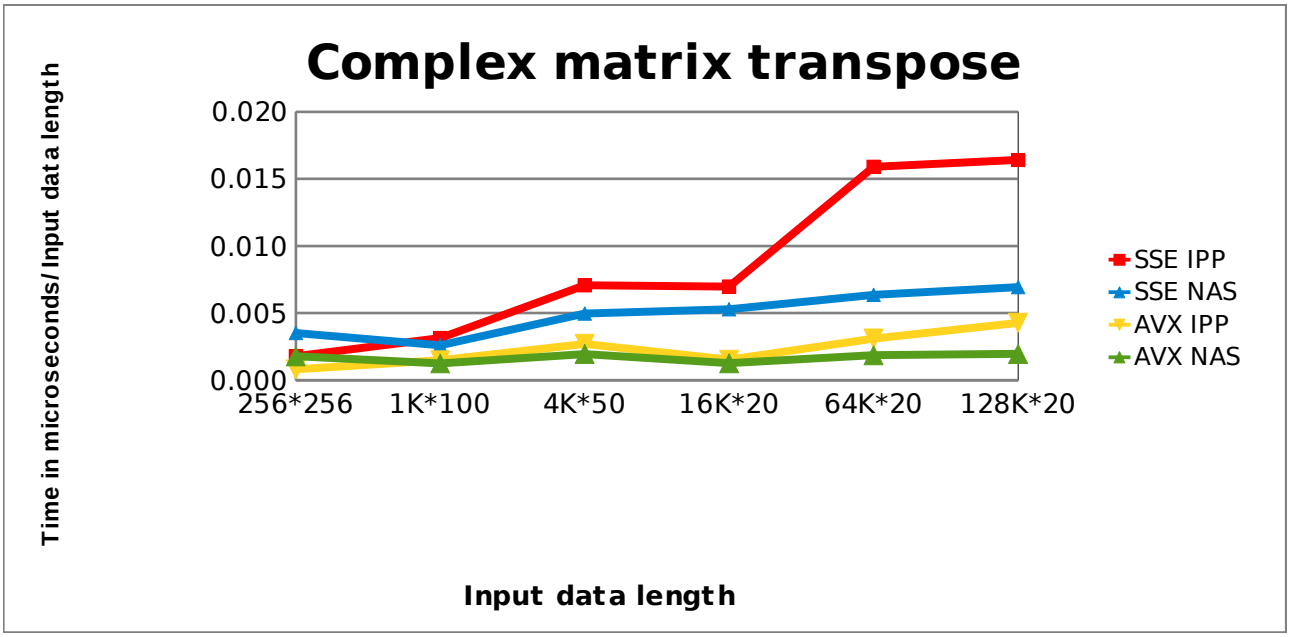
Table 26: vsip_cmtrans_f timings.

Input Data Length	256*256	1K*100	4K*50	16K*20	64K*20	128K*20
SSE IPP	117	321	1,446	2,283	20,837	42,994
SSE NAS	230	267	1,016	1,731	8,325	18,178
AVX IPP	53	151	554	498	4,063	11,217
AVX NAS	117	128	398	415	2,436	5,137

These timings show that IPP is better than NAS at 256 by 256. However, NAS is substantially better than IPP for the other data lengths. This is because NAS has an optimized algorithm for misaligned data within the transpose operation. The graph below shows the performance in time divided by data size. So the fastest algorithm is now the lowest line in the graph.

Figure 20: complex matrix transpose performance.

Graph shows time in microseconds / data size



In section 3 neither NAS or IPP showed a significant increase in performance between SSE and AVX. This was largely down to the fact that the operation is mainly memory dependent and is not computationally expensive. However, when we compare the Arrandale system to the AVX system there is a large performance advantage to AVX due to the faster memory speed. The table below shows the speed ups for all data sizes with IPP being 5.13 times quicker for AVX at a data length of 256K by 20:

Table 27: vsip_ccfftmip_f speed ups.

Input Data Length	256*256	1K*100	4K*50	16K*20	256K*20	512K*20
IPP Speedup Rate.	2.21	2.13	2.61	4.58	5.13	3.80
NAS Speedup Rate.	1.97	2.09	2.55	4.17	4.42	3.54

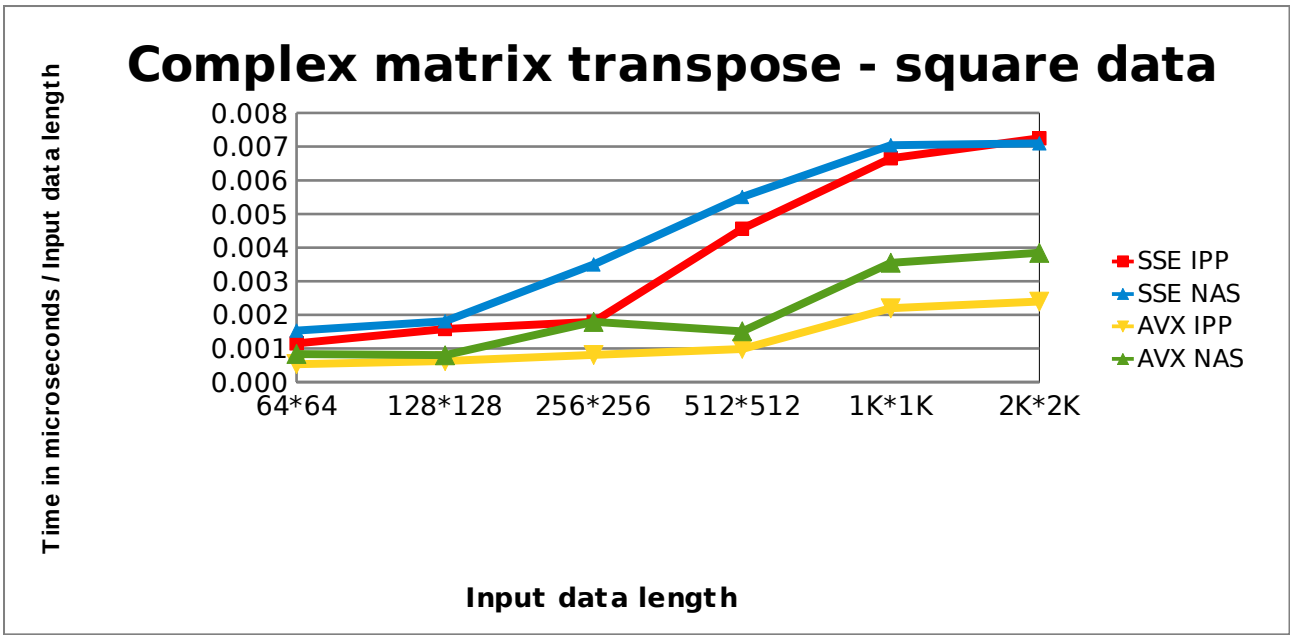
As with the 2d FFTs the efficiency of the algorithms are not as good as it would be with square matrices due to the data sizes being misaligned from row-to-row. The results also show a big difference in performance between NAS and IPP for both SSE and AVX with NAS being the most efficient choice for non-square matrices. The table below shows a series of square matrix times from a complex matrix of 64 by 64 cells up to 2048 by 2048 cells.

Table 28: vsip_cmtrans_f square matrix timings.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
SSE IPP	4.71	25.91	117.4	1,198	6,976	30,410
SSE NAS	6.27	29.67	229.9	1,443	7,381	29,762
AVX IPP	2.19	10.24	53.0	259	2,303	10,042
AVX NAS	3.43	13.10	117.2	395	3,720	16,118

Figure 21: complex matrix transpose performance.

Graph shows time in microseconds / data size



The above graphs show the execution time divided by the data size with fastest algorithms lower in the graphs. The slowest time is for SSE NAS shown in blue. However on square matrices the IPP AVX library is well optimized and ends up with the fastest algorithm shown in yellow. The IPP library has a good AVX performance across most of the data sizes considering the transpose operation is not computationally expensive. The table below shows the speed ups for all data lengths with AVX being 4.63 times quicker for IPP at a data length of 512 by 512:

Table 29: vsip_ccfftmip_f square matrix speed ups.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
IPP Speedup Rate.	2.15	2.53	2.22	4.63	3.03	3.03
NAS Speedup Rate.	1.83	2.26	1.96	3.65	1.98	1.85

4.5. Complex Vector Multiply.

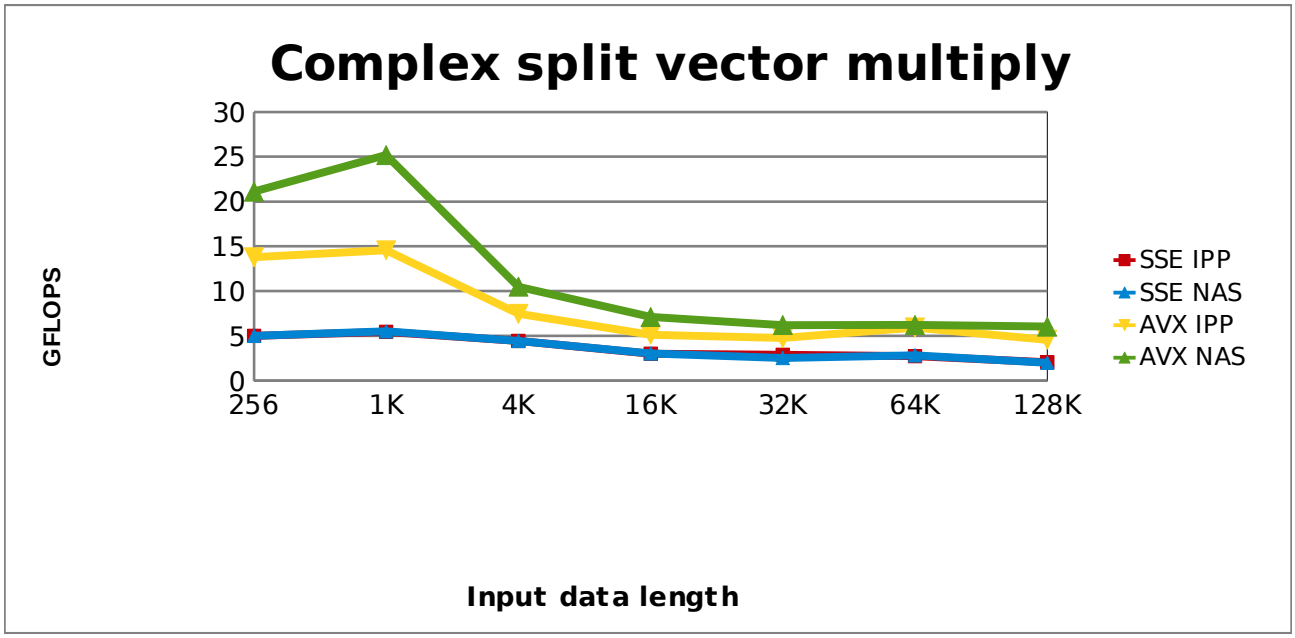
The following table and graph show the optimized SSE and AVX timings for the complex vector multiply operation. The table shows the results in microseconds and the graph has them in GFlops.

Table 30: vsip_cvmul_f timings.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP	0.30	1.10	5.39	31.95	66.45	140.5	374.1
SSE NAS	0.30	1.09	5.40	31.52	75.55	135.7	383.3
AVX IPP	0.11	0.42	3.29	19.18	41.30	66.2	172.1
AVX NAS	0.07	0.24	2.34	13.82	31.69	63.2	130.0

Figure 22: Split complex vector multiply performance.

MFLOPS = 6 * N / (time for one vector multiply in microseconds)



The red and blue graphs show that NAS and IPP algorithms have a similar performance on the Arrandale platform. Both NAS and IPP show a large increase in performance on the AVX platform with AVX optimized code. The peak performance is at 1K with 25 GFlops. The following table shows the rate of performance increase with the NAS library performing 4.53 times quicker at a data length of 1K.

Table 31: vsip_cvmul_f speed ups.

Input Data Length	256	1K	4K	16K	32K	64K	128K
IPP Speedup Rate.	2.73	2.62	1.64	1.67	1.61	2.12	2.17
NAS Speedup Rate.	4.28	4.53	2.31	2.28	2.38	2.15	2.95

4.6. Vector Sine.

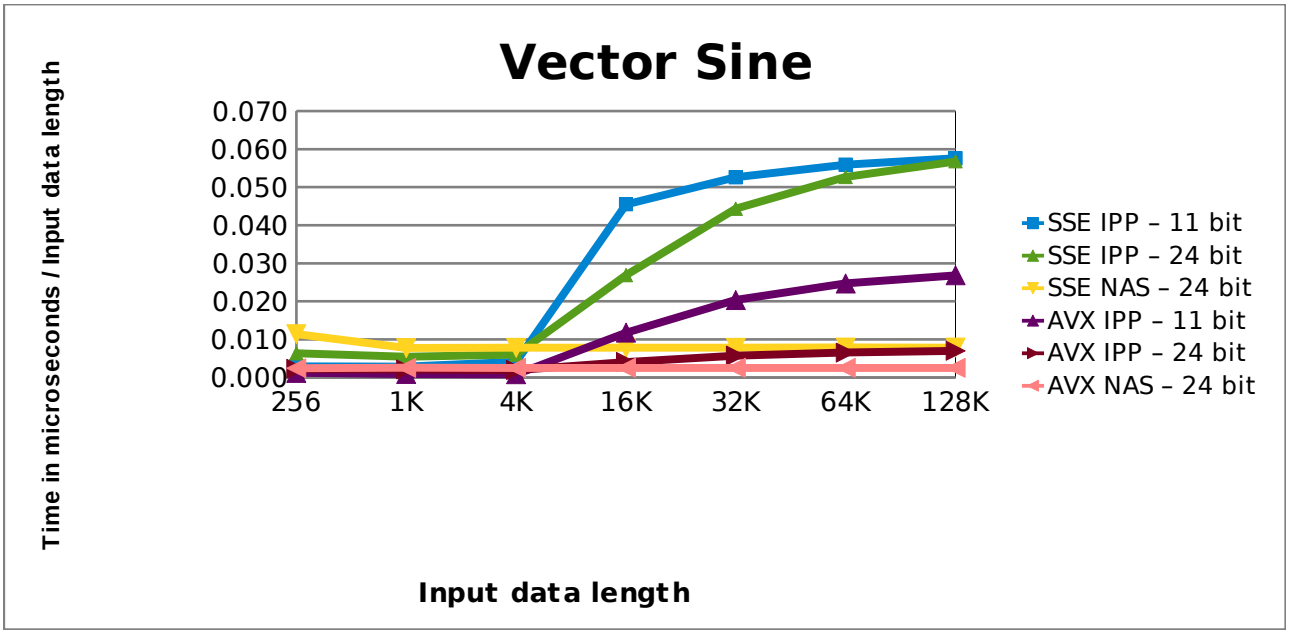
The following table and graph show the optimized SSE and AVX for the vector sin operation. The table has timings in microseconds and the graph displays the time in microseconds divided by the input data size. The IPP library has different floating point accuracies for this operation. The NAS call is always a full accuracy of 24 bits.

Table 32: vsip_vsin_f timings.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP – 11 bit	0.75	2.88	16.87	746	1725	3,666	7,554
SSE IPP – 24 bit	1.62	5.58	24.48	441	1453	3,455	7,449
SSE NAS – 24 bit	2.92	7.93	32.00	128	256	516	1,028
AVX IPP – 11 bit	0.31	0.91	3.34	193	667	1,618	3,518
AVX IPP – 24 bit	0.57	2.15	8.00	67	188	429	915
AVX NAS – 24 bit	0.62	2.61	10.20	41	82	163	327

Figure 23: vector sine performance.

Graph shows time in microseconds / data length



The yellow graph shows the NAS SSE time and the pink graph below shows the NAS AVX time. This has a good AVX performance across all data lengths. The algorithm for both NAS and IPP carries out a full range reduction to make sure the input values are in the correct quadrant. This algorithm was improved in the last release of the AVX IPP library. The IPP performance is good until we reach 4K and then the timings increase substantially. The table below shows the speed ups across all data lengths for IPP and NAS at 24 bit accuracy. The changes to the range reduction algorithm in the 24 bit call to IPP may explain the very large speed ups obtained at 128K. At a data length of 256 the NAS version is 4.7 times quicker on AVX.

Table 33: . vsip_vsin_f speed ups.

Input Data Length	256	1K	4K	16K	32K	64K	128K
IPP Speedup Rate.	2.80	2.59	3.06	6.50	7.73	8.05	8.14
NAS Speedup Rate.	4.70	3.04	3.14	3.12	3.12	3.17	3.14

4.7. Vector Cosine.

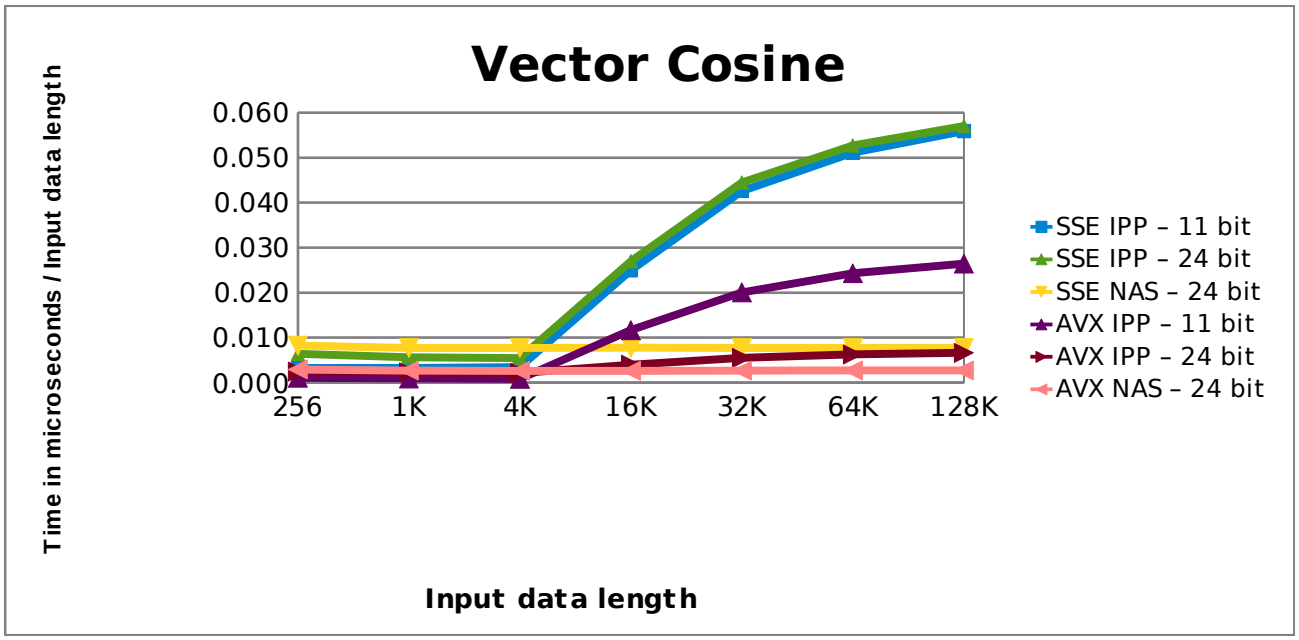
The following table and graph show the optimized SSE and AVX timings for the vector cosine operation. The table is in microseconds and the graph displays the time in microseconds divided by the input data size. In the data below we have given results for IPP operating with a low accuracy of 11 bits and both IPP/NAS calling high accuracy algorithms of 24 bits.

Table 34: vsip_vcos_f timings.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP – 11 bit	0.84	3.34	14.07	413	1,397	3,350	7,329
SSE IPP – 24 bit	1.65	5.76	22.44	445	1,457	3,458	7,477
SSE NAS – 24 bit	2.11	7.90	31.61	127	253	506	1,012
AVX IPP – 11 bit	0.28	0.98	3.57	191	658	1,594	3,468
AVX IPP – 24 bit	0.63	2.36	8.87	66	181	411	871
AVX NAS – 24 bit	0.75	2.75	10.87	43	88	177	355

Figure 24: vector cosine performance.

Graph shows time in microseconds / data length



The vector cosine operation has a similar graph to the vector sin operation. The AVX performance of the NAS cosine operation is fairly constant across all data lengths. As was the case with the sine test these results look very good given the lack of support of 8way SIMD integer operations within the AVX instruction set. The table below shows that at a data length of 256 the NAS algorithm is 4.7 times quicker on AVX. IPP is 8.14 times quicker at 128K but some of this could be down to changes in the algorithm.

Table 35: vsip_vcos_f speed ups.

Input Data Length	256	1K	4K	16K	32K	64K	128K
IPP Speedup Rate.	2.80	2.59	3.06	6.50	7.73	8.05	8.14
NAS Speedup Rate.	4.70	3.04	3.14	3.12	3.12	3.17	3.14

4.8. Vector Square Root.

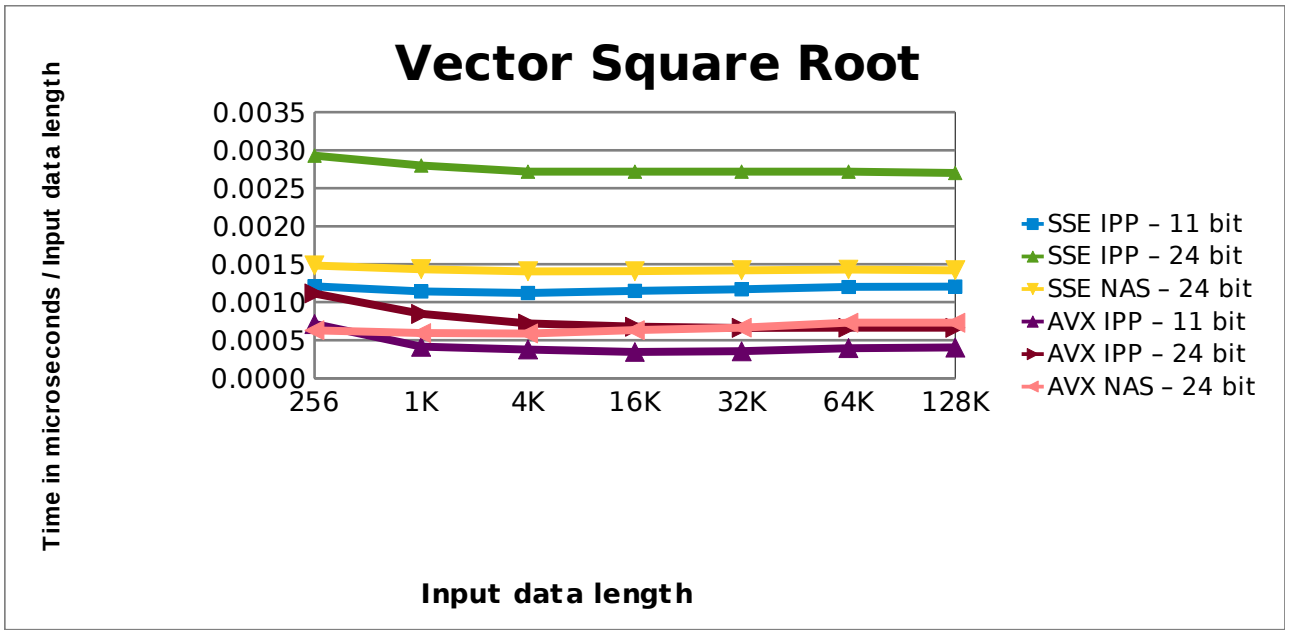
The following table and graph show the optimized SSE and AVX timings for the vector square root operation. The table is in microseconds and the graph displays the time in microseconds divided by the input data size. The NAS algorithm has a full floating point accuracy of 24 bits. The IPP library has different accuracies for the square root function. Accuracies 11 and 24 bits have been given in the table below.

Table 36: vsip_vsqrt_f timings.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP – 11 bit	0.31	1.17	4.59	18.85	38.36	78.93	158.11
SSE IPP – 24 bit	0.75	2.87	11.16	44.66	89.07	178.23	356.36
SSE NAS – 24 bit	0.38	1.47	5.76	23.07	46.67	93.80	187.25
AVX IPP – 11 bit	0.18	0.43	1.55	5.66	11.66	25.89	53.05
AVX IPP – 24 bit	0.29	0.87	2.95	11.11	21.70	43.58	87.01
AVX NAS – 24 bit	0.16	0.61	2.41	10.37	21.78	47.91	95.84

Figure 25: vector square root performance.

Graph shows time in microseconds / data length



The blue and purple graphs are for IPP 11 bit accuracy. These produce the fastest answer on the AVX machine but are not as accurate as the other graphs. The yellow and pink graphs represent the NAS square root operations with the yellow graph being the SSE performance and the pink graph representing the AVX performance. Here we get a good AVX performance across all data lengths. The NAS algorithm is 2.41 times quicker on AVX at a data length of 1K. The IPP function speeds up by as much as 4.1 times on AVX. However, the SSE version at 24 bit accuracy in the above graph produces the worst performance.

Table 37: vsip_vsqrt_f speed ups.

Input Data Length	256	1K	4K	16K	32K	64K	128K
IPP Speedup Rate.	2.59	3.29	3.78	4.02	4.10	4.09	4.10
NAS Speedup Rate.	2.38	2.41	2.39	2.22	2.15	1.95	1.95

4.9. Vector Scatter.

The vector scatter operation copies an input vector into an output vector using another input vector of indexes stating the position of each element in the output vector. In the tests below the output vector is twice the size of the input vector vector and the elements are scattered into every other output position. The data lengths below give the input vector size. The code in the SSE version was serial code because of the simple nature of the operation. The table below shows the serial Arrandale SSE time and the serial AVX time. The AVX compiler optimization produced a significant improvement in performance by just compiling the serial code. At this point NAS optimized the algorithm to take advantage of AVX 8way SIMD floating point load instructions given that the data in the input vectors are aligned in these tests. The last row in the table below shows this AVX optimized time. The timings in the table are given in microseconds.

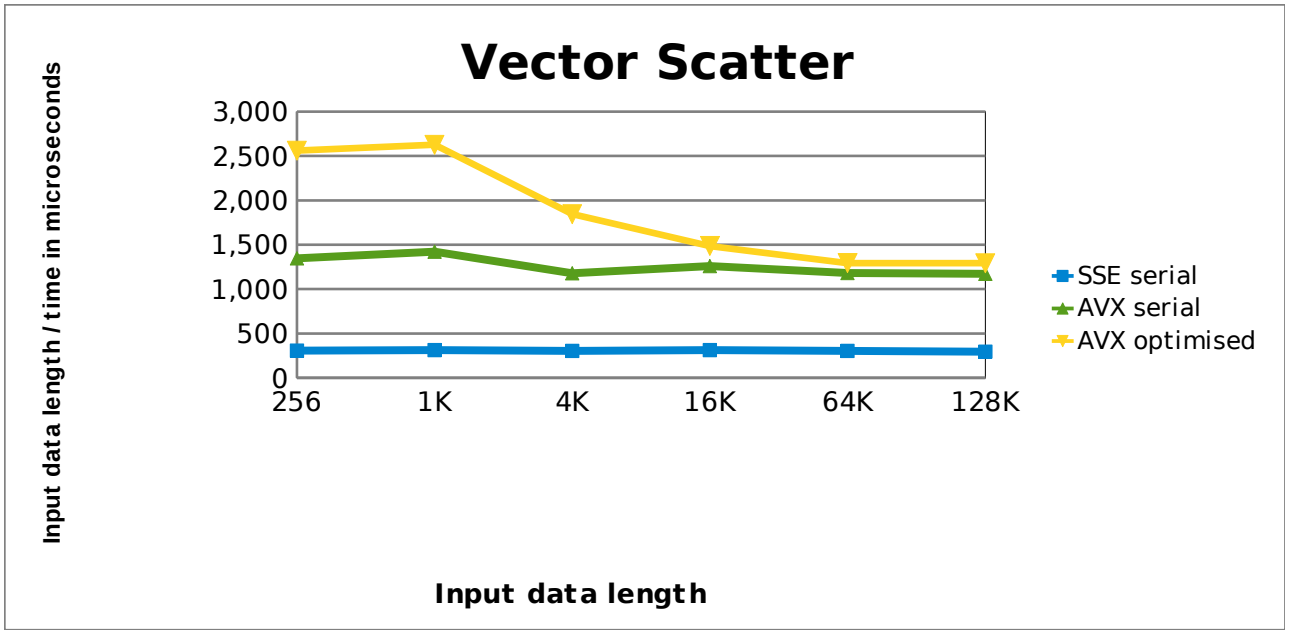
Table 38: vsip_vscatter_f timings.

Input Data Length	256	1K	4K	16K	64K	128K
SSE serial	0.83	3.28	13.45	52.47	214.47	441.32
AVX serial	0.19	0.72	3.48	13.00	55.50	111.75
AVX Optimized	0.1	0.39	2.22	11.03	50.69	101.52

The above timings show that the AVX serial code is roughly 4 times quicker on the AVX machine than it is on the Arrandale machine. Section 3.9 showed that by simply compiling this serial code with an AVX compiler gave us a speed up of around 1.6. The timings above show that this speed up grows to about 4 when comparing code ran on the Arrandale system. The AVX hand coded optimisations then produced another speed up 1.85 in performance at 1K. The following graph shows the input data size divided by the time in microseconds to highlight the performance of the algorithm on the AVX machine.

Figure 26: vector scatter performance.

Graph shows data length / time in microseconds



The table below shows that the AVX optimised code is 8.41 times quicker at 1K than the serial code on the Arrandale platform. The AVX serial code is 4.55 times quicker at 1K than the SSE equivalent.

Table 39: vsip_vscatter_f speed ups.

Input Data Length	256	1K	4K	16K	64K	128K
AVX serial speedup.	4.36	4.55	3.86	4.04	3.86	3.95
AVX optimize speedup.	8.30	8.41	6.05	4.76	4.23	4.35

4.10. Vector Gather.

The vector gather operation usually gathers elements from a large input vector into a smaller output vector. The smaller output vector comes with a vector of indexes stating the position of the elements in the larger vector. The output vector and vector of indexes are the same size and are given in the table below. The table shows the SSE serial time, the AVX serial time and the AVX optimised time.

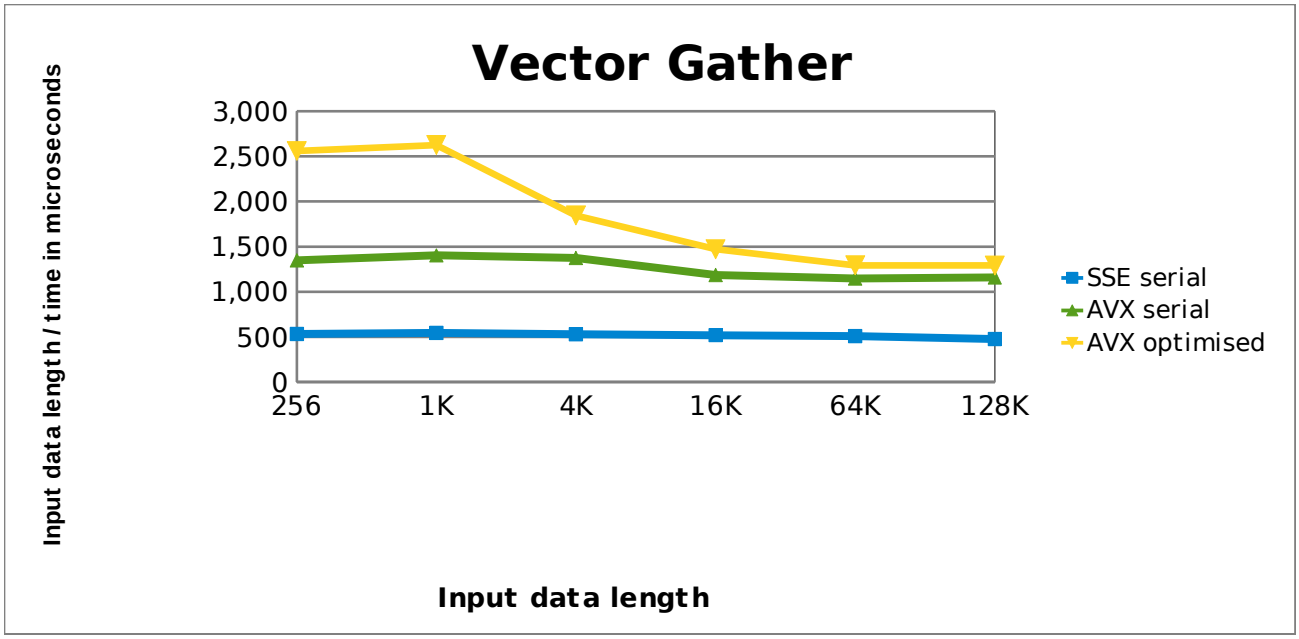
Table 40: vsip_vgather_f timings.

Input Data Length	256	1K	4K	16K	64K	128K
SSE serial	0.48	1.88	7.74	31.56	128.77	275.17
AVX serial	0.19	0.73	2.98	13.84	57.21	113.13
AVX optimized	0.1	0.39	2.22	11.14	50.7	101.49

The compiler optimisations are slightly less than the vector scatter test but are very still significant. The largest speed up from serial SSE code to optimised AVX code is at 4K and is 2.59. The graph below shows the input data size divided by the time in microseconds.

Figure 27: vector gather performance.

Graph shows data length / time in microseconds



The table below shows the speed ups on AVX for all data lengths. In the AVX optimized speed up we are comparing serial code on Arrandale to SIMD code on the AVX machine. The maximum speed up is at 1K where the optimized code runs 4.82 times quicker on AVX.

Table 41: vsip_vgather_f speed ups.

Input Data Length	256	1K	4K	16K	64K	128K
AVX serial speedup.	2.52	2.57	2.59	2.28	2.25	2.43
AVX optimize speedup.	4.80	4.82	3.49	2.83	2.54	2.71

5. The Compiler Performance.

The performance figures given in sections 3 and 4 were based on using the IPP library and a series of NAS AVX optimized routines written in C code. (Apart from the NAS FFTs which are written in assembler). The NAS C code contains vectorized algorithms and calls to built in functions that perform 8-way SIMD operations. These DSP algorithms are accessed via VSIPL wrappers also written in C code. The IPP library is a compiled library that is pulled in at link time. The wrappers and the NAS DSP algorithms were compiled with gcc version 4.4.6 to obtain the results presented in the last section. In this section we examine the results of compiling the C code with Intel's icc compiler. To do this we have used icc version 12.

It did not make any difference to the speed of the algorithms if the VSIPL wrapper was compiled with gcc or icc. The same timings were obtained for both compilers. However, it sometimes made a significant difference which compiler was used to compile the NAS AVX optimized routines. These routines consist of C code that have calls to built-in functions accessing the Single Instruction Multiple Data (SIMD) features of the chip. These SIMD instructions take advantage that the registers are twice the width (256 bits) on the AVX platform. The following subsections show the performance obtained using gcc and icc on a number of NAS DSP operations.

5.1. The NAS Transpose Operation.

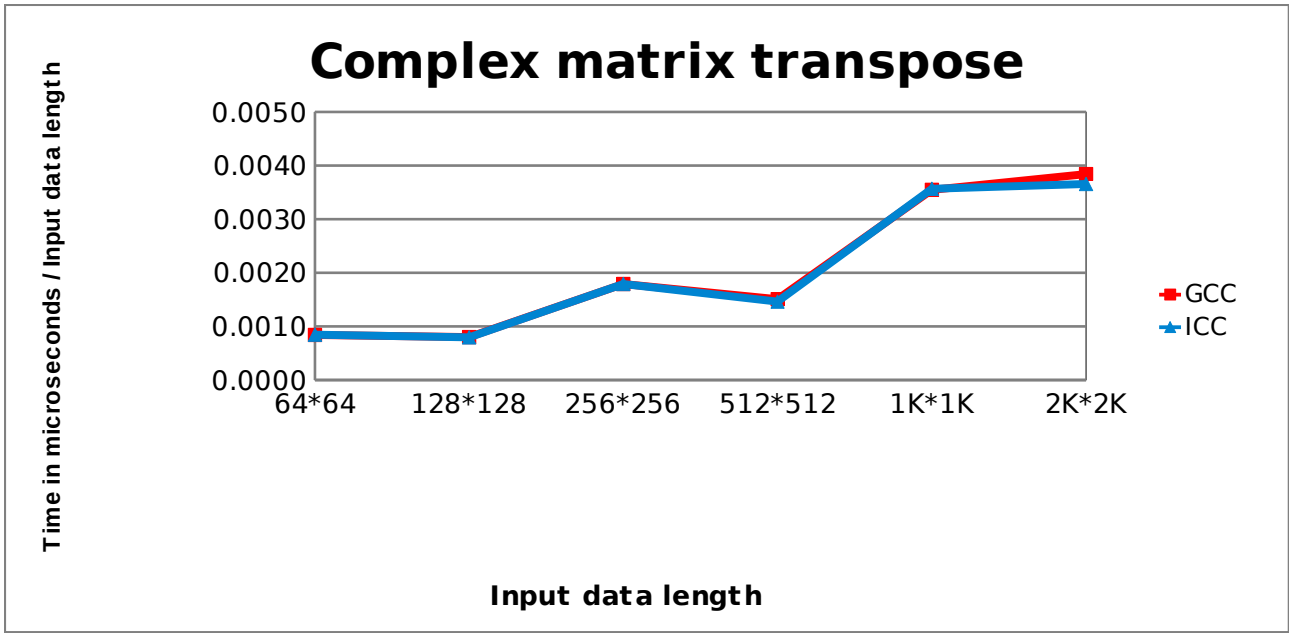
The following table shows the time obtained from the NAS transpose operation when compiled with gcc and icc over square matrices:

Table 42: vsip_cmtrans_f timings for gcc and icc.

Input Data Length	64*64	128*128	256*256	512*512	1K*1K	2K*2K
GCC	3.43	13.10	117.2	395	3,720	16,118
ICC	3.46	13.01	117.4	384	3,740	15,330

Figure 28: vsip_cmtrans_f performance for gcc and icc.

Graph shows time in microseconds / data length



As can be seen in the graph above both compilers gave the same performance apart from 2K by 2K where the icc compiler was 4.9% quicker.

5.2. The NAS AVX Complex Vector Multiply Operation.

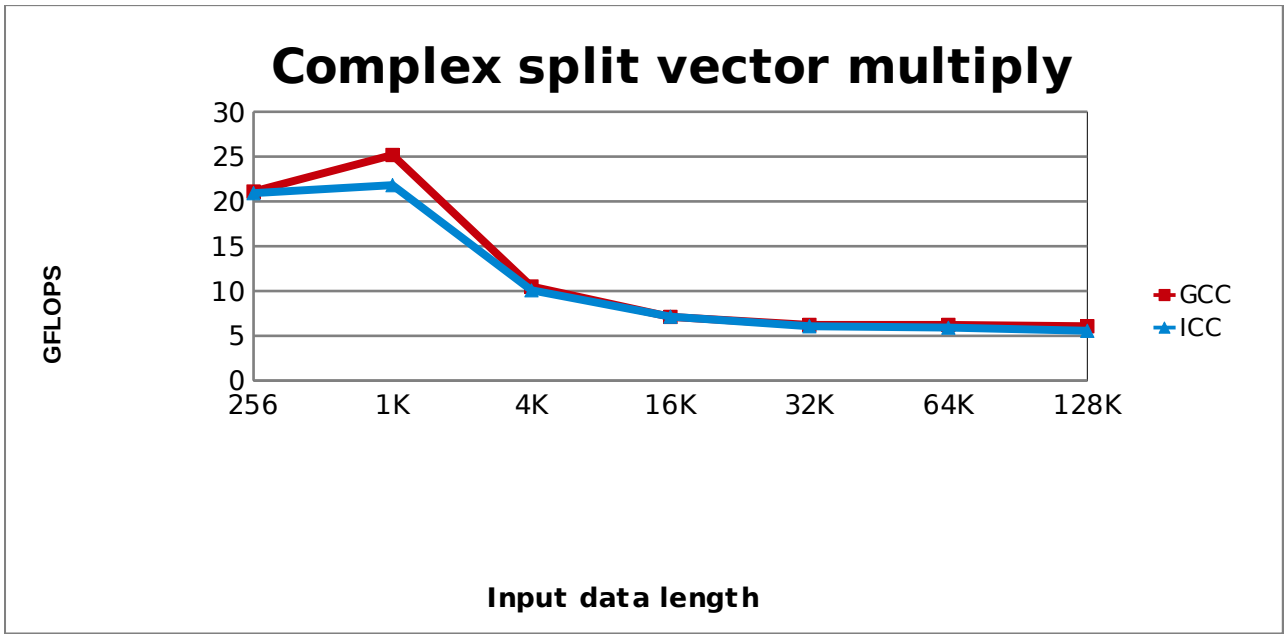
The following table shows the time obtained from the NAS complex split multiply operation when compiled with gcc and icc over a specified range of vectors. The table is in microseconds and the graph is in GFlops.

Table 43: vsip_cvmul_f timings for gcc and icc.

Input Data Length	256	1K	4K	16K	32K	64K	128K
GCC	0.07	0.24	2.34	13.82	31.69	63.23	129.99
ICC	0.07	0.28	2.44	13.77	32.42	66.66	141.80

Figure 29: vsip_cvmul_f performance for gcc and icc.

MFLOPS = 6 * N / (time for one vector multiply in microseconds)



Both compilers gave similar results except for the 1K data length where this time the gcc compiler was quicker by 14%.

5.3. The NAS AVX Sine Operation.

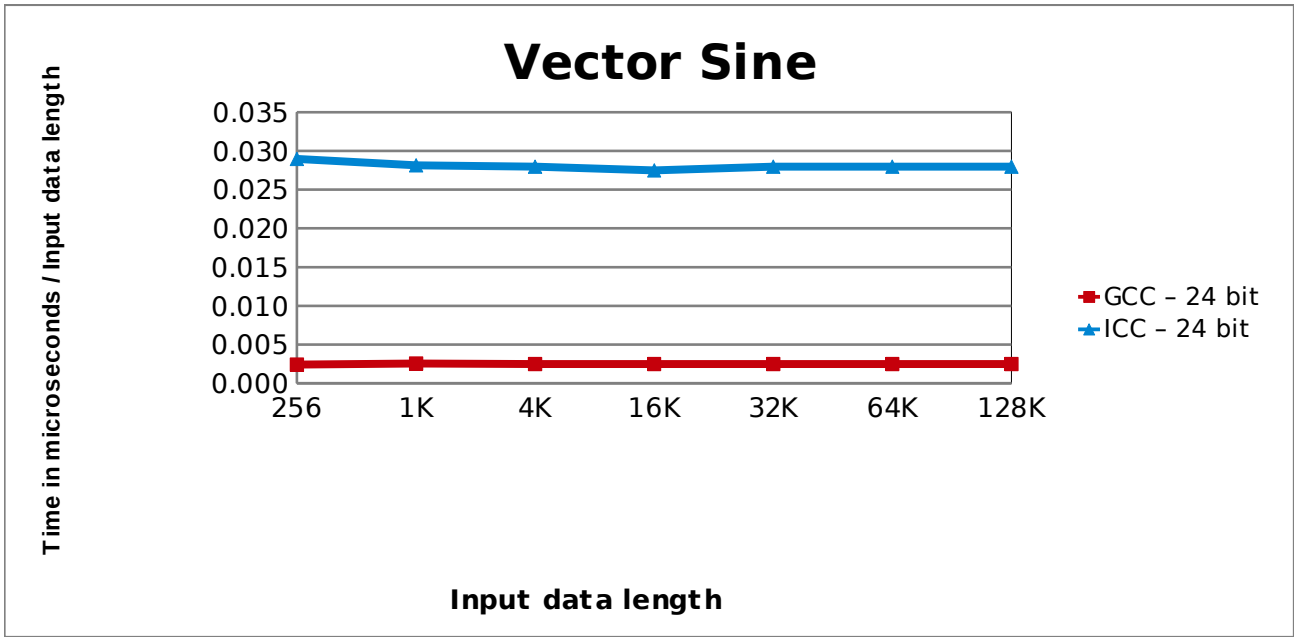
The following table shows the time obtained from the NAS AVX Sine operation when compiled with gcc and icc over a specified range of vectors. The times in the table are given in microseconds.

Table 44: vsip_vsin_f timings for gcc and icc.

Input Data Length	256	1K	4K	16K	32K	64K	128K
GCC – 24 bit	0.62	2.61	10.20	40.93	81.53	163.36	326.94
ICC – 24 bit	7.42	28.83	114.60	450.47	916.65	1,833.42	3,666.78

The following graph shows the huge difference in performance between gcc and icc, with gcc having a much better performance than icc for the sin code.

Figure 30: vector sin performance with icc and gcc.
Graph shows time in microseconds / data length



5.4. The NAS AVX Cosine Operation.

The following table shows the time obtained from the NAS AVX Cosine operation when compiled with gcc and icc over a specified range of vectors. The times in the table are given in microseconds.

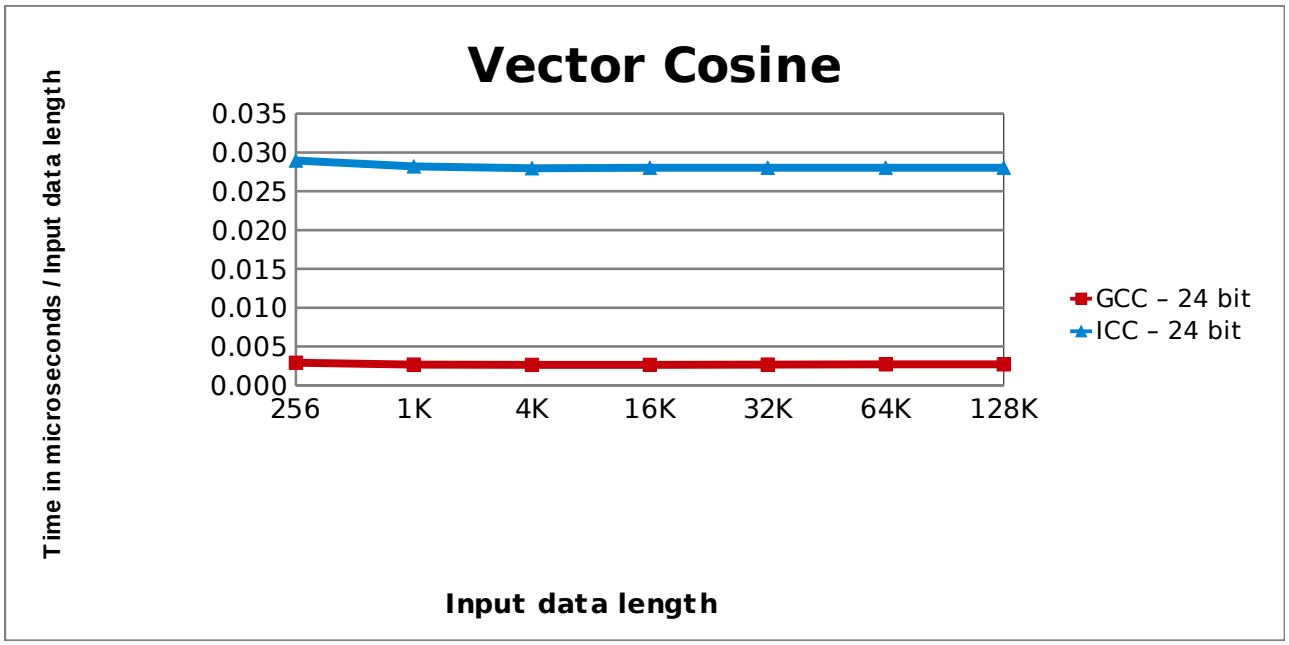
Table 45: vsip_vcos_f timings for gcc and icc.

Input Data Length	256.00	1K	4K	16K	32K	64K	128K
GCC – 24 bit	0.75	2.75	10.87	43.46	87.79	177.38	354.83
ICC – 24 bit	7.41	28.89	114.61	459.20	919.01	1,837.82	3,675.69

The timings are very similar to the sin results in the previous subsection. The icc compiler timings are substantially higher than the gcc timings for all data lengths. The following graph also highlights this performance difference.

Figure 31: vector cosine performance with icc and gcc.

Graph shows time in microseconds / data length



5.5. The NAS AVX Square Root Operation.

The following table shows the time obtained from the NAS AVX square root operation when compiled with gcc and icc over a specified range of vectors. The times in the table are given in microseconds.

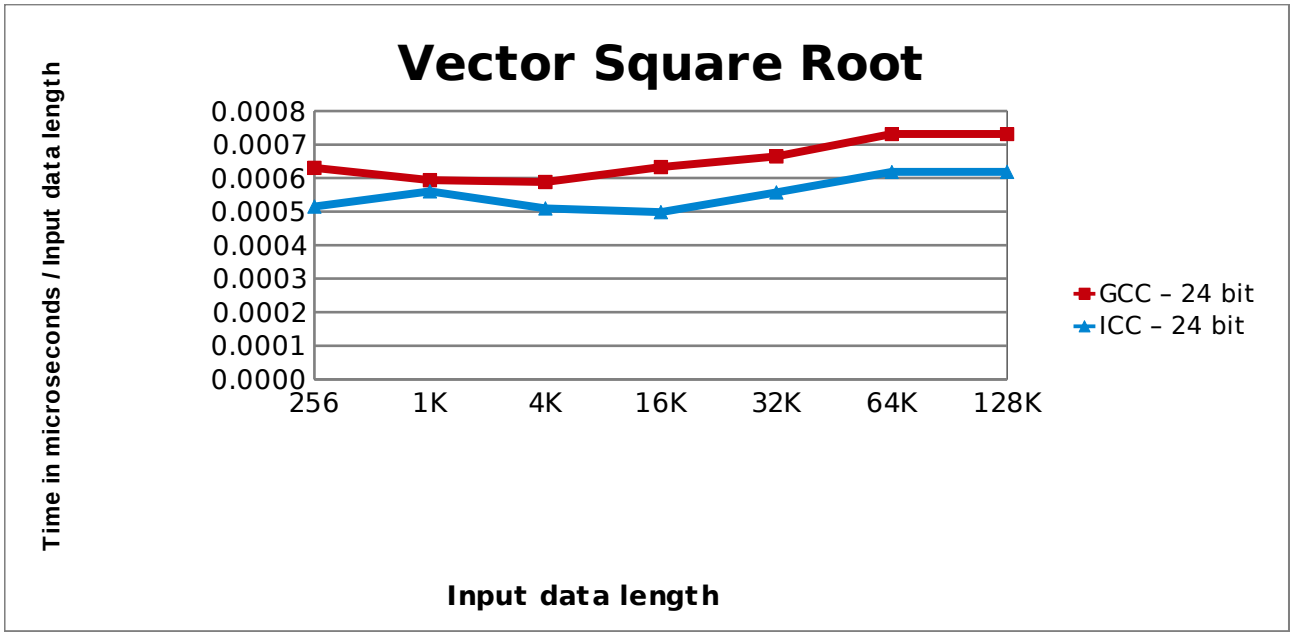
Table 46: vsip_vsqrt_f timings for gcc and icc.

Input Data Length	256.00	1K	4K	16K	32K	64K	128K
GCC – 24 bit	0.16	0.61	2.41	10.37	21.78	47.91	95.84
ICC – 24 bit	0.13	0.57	2.09	8.16	18.25	40.51	81.07

The above table shows that this time the icc compiler is faster than the gcc compiler for all data lengths. The performance improvement for icc varies from 21% down to 6% depending on the vector length. The following graph highlights the improved performance obtained by icc for the vector square root operation.

Figure 32: vector square root performance with icc and gcc.

Graph shows time in microseconds / data length



5.6. The NAS AVX Scatter Operation.

The following table shows the time obtained from the NAS AVX vector scatter operation when compiled with gcc and icc over a specified range of vectors. The times in the table are given in microseconds.

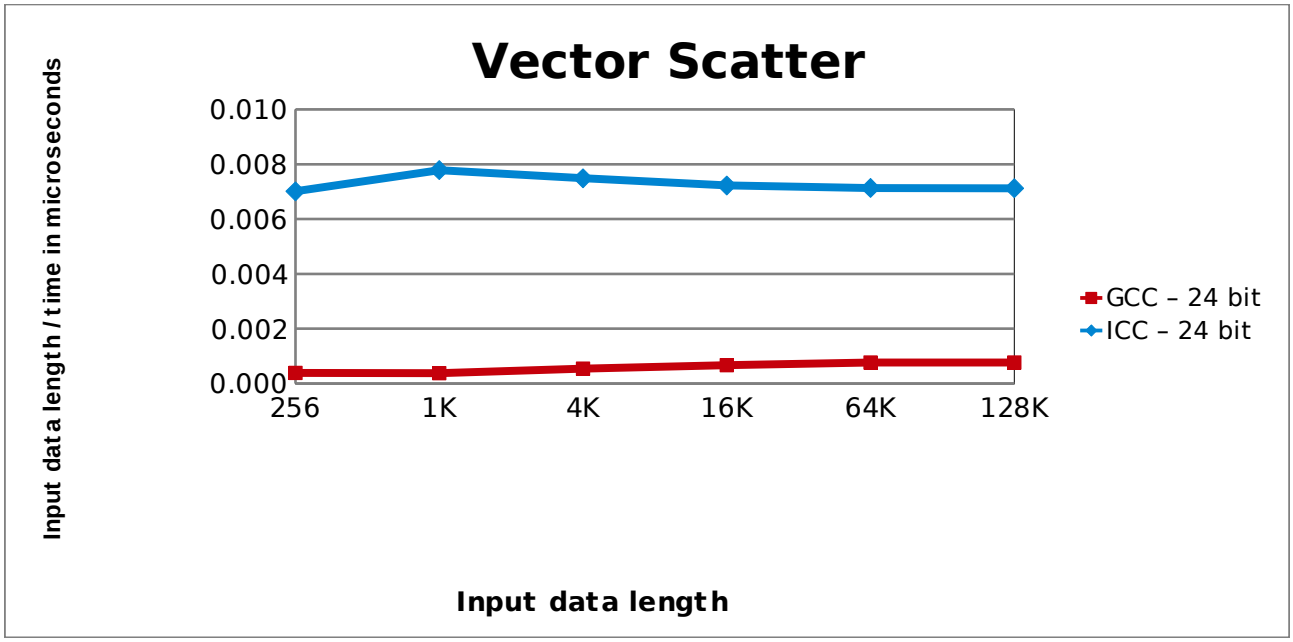
Table 47: vsip_vscatter_f timings for gcc and icc.

Input Data Length	256.00	1K	4K	16K	64K	128K
GCC – 24 bit	0.1	0.39	2.22	11.03	50.69	101.52
ICC – 24 bit	1.8	7.97	30.67	118.39	467.59	933.79

The above table shows that the gcc compiler produces the fastest times for all data lengths. The timings vary between between gcc being 90% and 95% faster for this operation as highlighted with the following graph.

Figure 33: vector scatter performance with icc and gcc.

Graph shows data length / time in microseconds



5.7. The NAS AVX Gather Operation.

The following table shows the time obtained from the NAS AVX vector gather operation when compiled with gcc and icc over a specified range of vectors. The times in the table are given in microseconds.

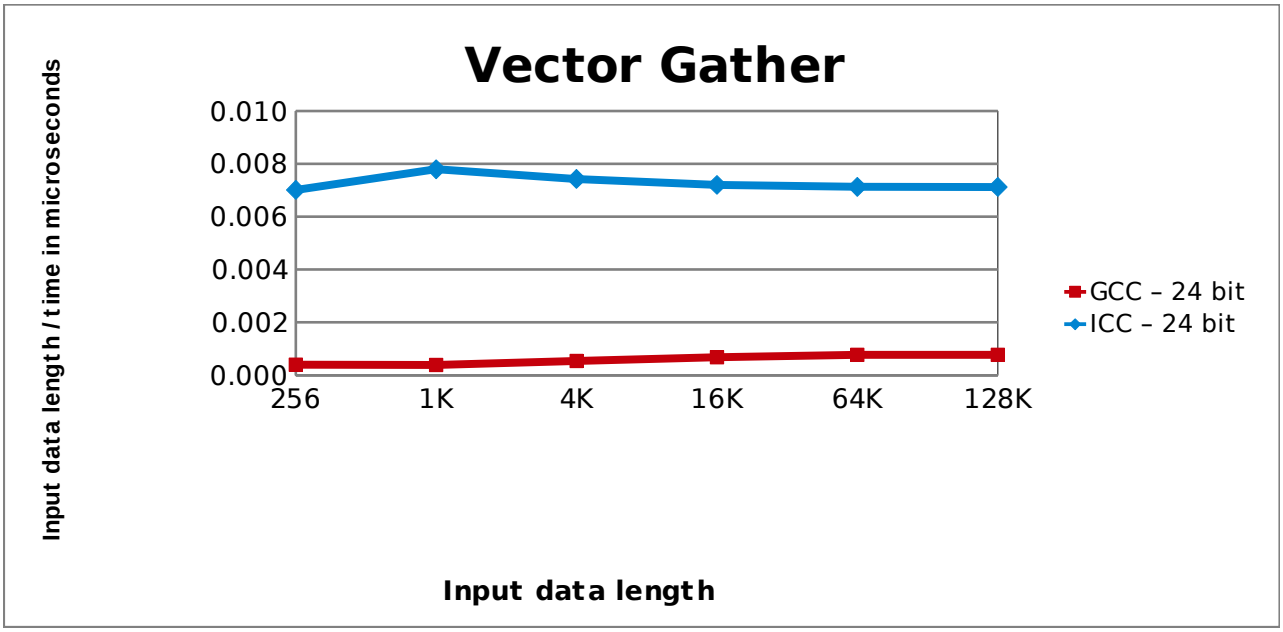
Table 48: vsip_vgather_f timings for gcc and icc.

Input Data Length	256.00	1K	4K	16K	64K	128K
GCC – 24 bit	0.10	0.39	2.22	11.14	50.70	101.49
ICC – 24 bit	1.80	7.98	30.43	118.03	467.41	933.75

The above results are very similar to the vector scatter results with gcc having a much higher performance with the vector gather operation. The following graph also highlights this performance.

Figure 34: vector gather performance with icc and gcc.

Graph shows data length / time in microseconds



5.8. Compiler Summary.

When studying which compiler to use between icc and gcc with the NAS AVX code we concluded the following:

Table 49: Optimal compiler summary.

Set of NAS AVX operations.	Optimal Compiler choice.	Speedup.
Complex matrix transpose. Complex vector multiply.	Either gcc or icc.	0
Vector Sin. Vector Cos. Vector Scatter. Vector Gather.	gcc.	90%+
Vector square Root.	icc	6% to 21%

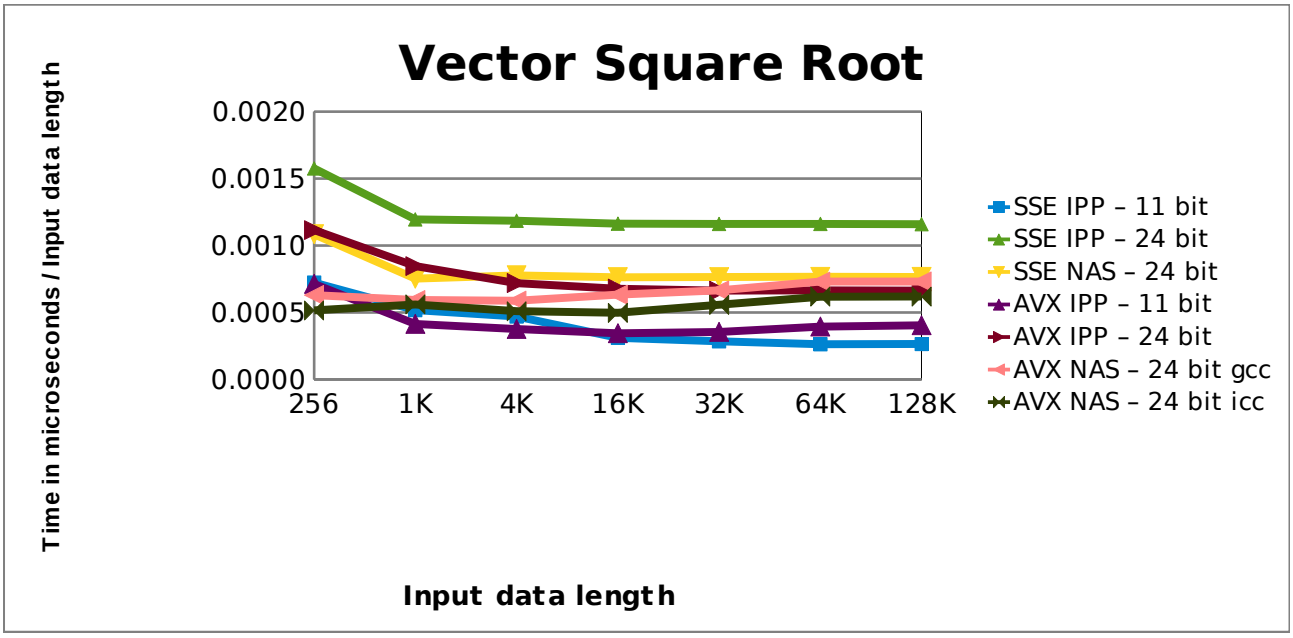
Therefore, the only operation in section 3 that would be significantly improved by using the icc compiler is the vector square root operation. The following table/graph shows how this would effect the SSE/AVS/NAS/IPP comparison:

Table 50: vsip_vsqrt_f timings with icc.

Input Data Length	256	1K	4K	16K	32K	64K	128K
SSE IPP – 11 bit	0.18	0.53	1.92	5.11	9.32	17.24	34.63
SSE IPP – 24 bit	0.40	1.22	4.85	19.06	38.02	76.13	151.84
SSE NAS – 24 bit	0.28	0.77	3.19	12.50	25.04	50.21	100.28
AVX IPP – 11 bit	0.18	0.43	1.55	5.66	11.66	25.89	53.05
AVX IPP – 24 bit	0.29	0.87	2.95	11.11	21.70	43.58	87.01
AVX NAS – 24 bit gcc	0.16	0.61	2.41	10.37	21.78	47.91	95.84
AVX NAS – 24 bit icc	0.13	0.57	2.09	8.16	18.25	40.51	81.07

Figure 35: vector square root performance with icc.

Graph shows time in microseconds / data length



6. Conclusions.

The results presented in this report show a very significant performance gain when using AVX over SSE. We have compared running optimized SSE and AVX programs on the same AVX test platform and we have compared an SSE Arrandale platform to the AVX test platform by running the test programs on separate machines.

When the programs are ran on the same AVX machine the AVX optimized program uses all 256 bits of the machines registers and the richer AVX instruction set. The SSE optimized code uses the SSE instruction set and the first 128 bits of each register. This means the SSE code implements 4-way SIMD and the AVX code implements 8-way SIMD. When going from 4-way to 8-way SIMD the maximum performance gain you would expect is a doubling of the program speed. However, in the real world this is usually a lot less depending on cacheing issues. Indeed if programs are completely memory dependent then there would be no speed up. The table below shows the maximum performance gain using AVX over the data lengths that this study examined:

Table 51: SSE and AVX code running on the same AVX platform.

Operation.	AVX Maximum Speed Up.	
	Percentage	Ratio (Times faster)
1D in-place complex-to-complex FFT.	87%	1.77
2D in-place complex-to-complex FFT.	67%	1.50
2D FFTs with square matrices.	79%	1.65
Multiple in-place complex-to-complex FFT.	96%	1.92
Complex Matrix Transpose.	27%	1.16
Transpose with square matrices.	53%	1.36
Complex Vector Multiply.	93%	1.87
Vector Sine.	75%	1.60
Vector Cosine.	61%	1.44
Vector Square Root.	86%	1.75
Vector Scatter.	92%	1.85
Vector Gather.	93%	1.87

In the above table 100% represents a doubling of the program speed and the above percentages are the maximum gain, which usually occurs for smaller data lengths with less cacheing issues. As the data lengths become larger cache issues then become more significant and eventually the algorithm is more memory dependent than processing dependent. When this occurs the performance gains will be a lot less.

The above tables show the advantage of the wider registers on the AVX platform giving the AVX optimized program more processing power. The AVX platform has an improved memory controller and two 128-bit load ports giving the machine high memory speeds. The high memory speeds together with the the increased processing power of the AVX platform means a huge increase in performance of DSP algorithms over all data lengths. This can be seem when we compare the AVX optimized programs running on the AVX platform to SSE programs running on an SSE Arrandale system. This huge increase in performance was highlighted in detail in section 4 of this report. In defense/DSP applications such as SAR, Sonar, MTI,

Image processing and Medical scanning the algorithm is quite often memory bound due to the large amounts of data being processed. This means the AVX platform is particularly well suited to the task due to the improved CPU and memory performance. The table below shows the maximum speed up for the study operations over all study data lengths when comparing AVX to Arrandale:

Table 52: SSE4 Arrandale compared to and AVX platform.

Operation.	AVX Maximum Speed Up (Ratio).
1D in-place complex-to-complex FFT.	3.82
2D in-place complex-to-complex FFT.	3.06
2D FFTs with square matrices.	3.64
Multiple in-place complex-to-complex FFT.	3.67
Complex Matrix Transpose.	5.13
Transpose with square matrices.	4.63
Complex Vector Multiply.	4.53
Vector Sine.	4.70
Vector Cosine.	4.70
Vector Square Root.	4.10
Vector Scatter.	4.55
Vector Gather.	4.82

The above table shows that the AVX machine can be 3, 4 or even 5 times quicker than the Arrandale platform for these benchmarks. The above figures are the maximum speed up over all data lengths. These benchmarks were produced with a test program carrying out only this operation on the same buffer 100 times. The average timing is then calculated and shown in this report in sections 3 and 4. In an application when the operation is called with different blocks of memory and between lots of other DSP calls the speed ups will not be as high. In tests on applications such as SAR and MTI processors that have been optimized for AVX we have seen the AVX program running between 1.5 and 3 times quicker on the AVX machine.

When this study was first carried at the beginning of 2010 the IPP library had a poor performance with all FFTs using AVX. This report has now shown that IPP has a very good performance with 1D/2D FFTs and complex square transpose operations. The NAS algorithms were particularly good at vector multiply and sin/cosine/sqrt operations given the lack of support in AVX for 8way SIMD integer instructions. Both libraries show room for improvement and further work when processing non-square matrices of the dimensions that have been discussed.

The report also looked at the compiler choice between the gcc compiler and the icc compiler. The square root algorithm favored the icc compiler with a performance gain of between 6% and 21%. However, the sine, cosine, scatter and gather operations suffered a significant deterioration in performance using icc over gcc. Other operations studies in this report were neutral.

NA Software plan a new release of our AVX library in early 2011. This library will have an increased level of performance for the FFT algorithms as well as a range of other performance improvements. We plan to redo these comparisons at that point.