Cedric Blake
Ming Tai Ha
Vineet Sepaha
Allen Tung
Group3
11/22/2016
Parallel Computing

Project 2 - Newton's Method

ISPC Implementation

The main function of the ispc version essentially differs only in the function call to compute the square root. Rather than calling a sequential function, an external ISPC function is called, defined by newton.ispc. In this external file, we have an external function that creates a number of tasks equivalent to the 'cores' input argument specified by the user. The tasks are begun with the ispc launch command, and the task is also defined in newton.ispc. Input arguments to the task include an input array, an output array, the number of elements in the array, the number of cores desired, and the number of threads desired. In each task, one has access to the taskIndex and taskCount variables, which may be used to determine what indices of the array to process. The user defined number of threads are launched to process data in parallel; with ISPC intrinsics, one thread may safely be launched for each element in the array. The user should only define a number of threads lower than the size of the matrix for defined operation. The user may specify a negative number to launch one thread for each element. If the user only wishes to use one core, the launching mechanism is bypassed entirely, and the task is called directly.

tasksys.cpp from the example files included in the ISPC folder was compiled and linked in order to use the launch functionality of ISPC.

Using the foreach construct of ISPC should thread the process, allowing it to run in a SIMD fashion. This will achieve a speedup on the square root program, because each core has a vector SIMD unit that can run on instruction over multiple pieces of data simultaneously. Writing the program as such allows us to take advantage of the SIMD unit of one core, and thereby speed up the program. Theoretically, running the program on multiple cores would allow us to exploit more SIMD units, and run more instructions concurrently. It follows that the program would execute in less time if it had more processing units.

Each launched task would allow the system to take advantage of another core to run the tasks simultaneously, decreasing runtime as long as long as there are more cores available. Once the number of tasks is the same as the number of cores, increasing the number of tasks should only serve to slightly worsen performance, because of the overhead in creation and synchronization. As for threads, small numbers of threads should worsen performance, as there is both overhead in creation, and the SIMD properties of the core cannot be utilized fully. We expect the best

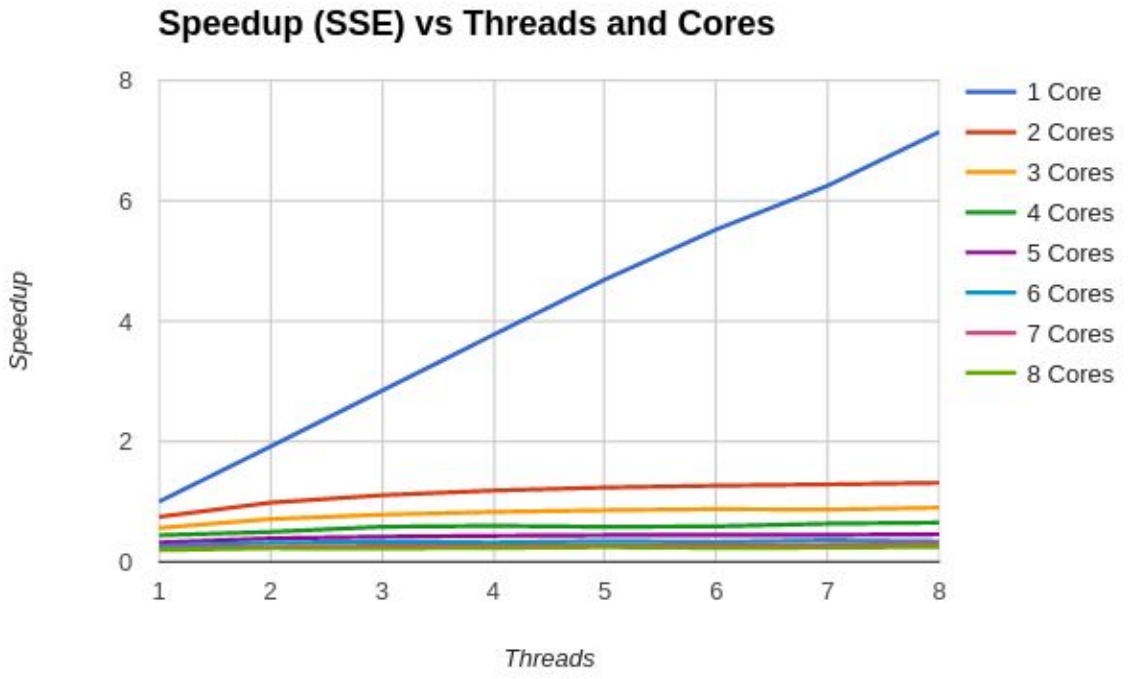
performance to be achieved when there is one ‘thread’ for each element in the array, allowing the foreach construct to take over the implementation and parallelize the threads.

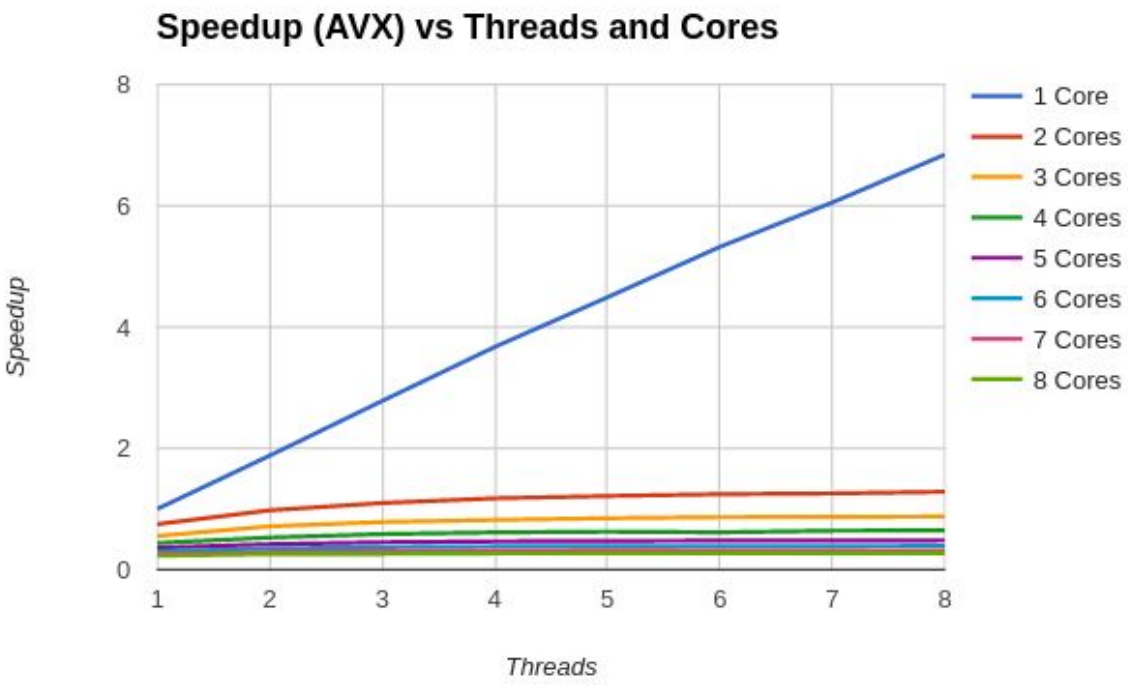
If we launch one thread for every single element of the array, we achieve a strong speedup as shown below in the Cores and Speedup graph. This is congruent with the hypothesis that utilizing the SIMD capabilities of a core will increase performance. If we launch much fewer threads, the full capabilities of the SIMD units cannot be utilized, and there is a much smaller speedup. Thus, when we increase the number of threads while it is very low, we observe a large increase in speedup. More threads executing would allow more usage of the SIMD architecture.

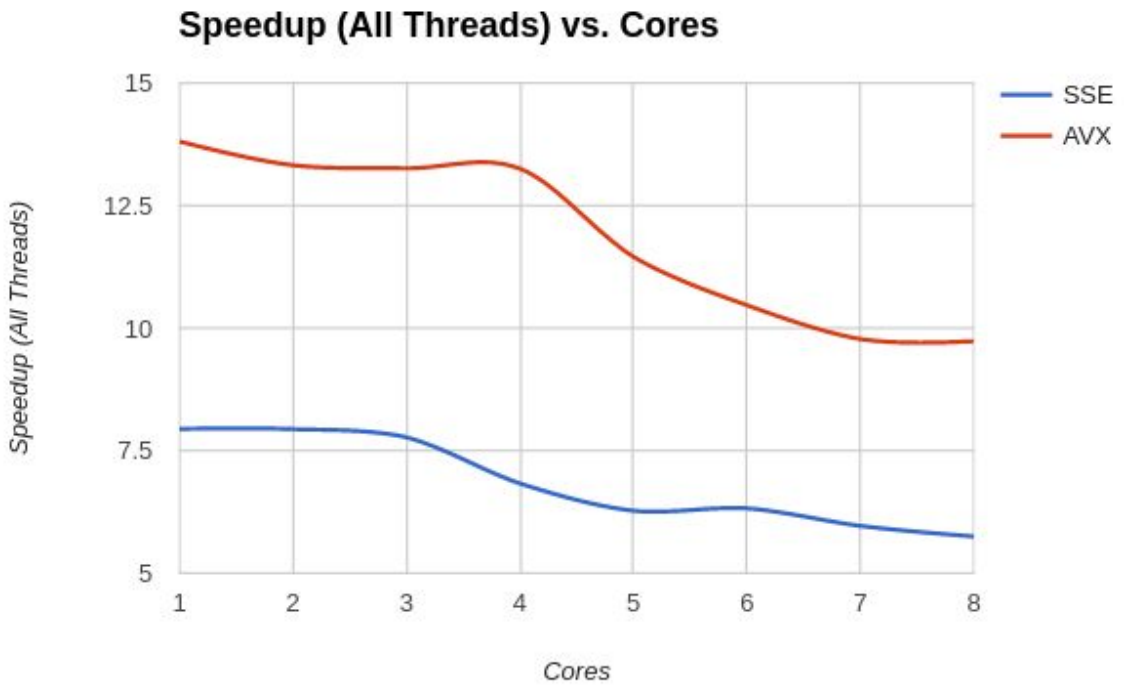
However, we do not produce larges speedup when increasing the number of cores with any number of threads. This may be explained with array accesses when using multiple cores. This particular implementation necessitates gather and scatter for loads and stores, which produces significant overhead with each core. The overhead for these gather and scatter instructions compounds to slightly worsen performance, as shown in the Cores and Speedup graph.

If full advantage is taken of the ISPC foreach construct, great speedup can be attained without the need of launching many additional tasks to utilize additional cores. However, if there are few threads, increasing the number of cores active at a time will achieve some speedups as well. This speedup, however, is offset by the gather and scatter overhead in memory access. If there are more tasks launched than cores, the system will perform slightly worse. The maximum speedup is attained when there is one thread launched for each element of the matrix, and there is one task launched for each core of the machine.

The following plots show the speedup per core per thread for SSE and AVX instructions, the speedup per core for SSE and AVX when all threads are used, and the raw data. The raw data can also be found in the code directory.


Speedup (SSE) vs Threads and Cores
Speedup
Threads
0
2
4
6
8
1
2
3
4
5
6
7
8
1 Core
2 Cores
3 Cores
4 Cores
5 Cores
6 Cores
7 Cores
8 Cores


Speedup (AVX) vs Threads and Cores
Speedup
Threads
0
2
4
6
8
1
2
3
4
5
6
7
8
1 Core
2 Cores
3 Cores
4 Cores
5 Cores
6 Cores
7 Cores
8 Cores



Cores	Threads	Time Taken	Speedup target SSE	Cores	Threads	Time Taken	Speedup target SSE
1	1	2.126385	1	5	1	6.629456	0.320748
1	2	1.109316	1.916843	5	2	5.44643	0.390418
1	3	0.747507	2.844636	5	3	5.094524	0.417386
1	4	0.563417	3.774087	5	4	4.906264	0.433402
1	5	0.453186	4.69208	5	5	4.794529	0.443502
1	6	0.384946	5.523853	5	6	4.741307	0.448481
1	7	0.340118	6.251904	5	7	4.754922	0.447197
1	8	0.297639	7.144175	5	8	4.638571	0.458414
1	-1	0.267733	7.942185	5	-1	0.339028	6.272004
2	1	2.842348	0.748109	6	1	7.833146	0.27146
2	2	2.161806	0.983615	6	2	6.716634	0.316585

2	3	1.920456	1.107229	6	3	6.332232	0.335803
2	4	1.794389	1.185019	6	4	6.596032	0.322373
2	5	1.720797	1.235698	6	5	6.14604	0.345976
2	6	1.678651	1.266723	6	6	6.424061	0.331003
2	7	1.652315	1.286913	6	7	5.920803	0.359138
2	8	1.619436	1.31304	6	8	6.281443	0.338519
2	-1	0.267884	7.937708	6	-1	0.336406	6.320889
3	1	3.774105	0.563414	7	1	9.415866	0.22583
3	2	2.990708	0.710997	7	2	8.503483	0.25006
3	3	2.710632	0.784461	7	3	8.085795	0.262978
3	4	2.563565	0.829464	7	4	7.981909	0.266401
3	5	2.479154	0.857706	7	5	7.846838	0.270986
3	6	2.425674	0.876616	7	6	7.523658	0.282626
3	7	2.449777	0.867991	7	7	7.489097	0.283931
3	8	2.362246	0.900154	7	8	6.980137	0.304634
3	-1	0.273663	7.770086	7	-1	0.356622	5.962574
4	1	4.815333	0.441586	8	1	10.939116	0.194384
4	2	4.274357	0.497475	8	2	9.378413	0.226732
4	3	3.648056	0.582882	8	3	9.497304	0.223894
4	4	3.551006	0.598812	8	4	9.232349	0.230319
4	5	3.656765	0.581493	8	5	8.696968	0.244497
4	6	3.602295	0.590286	8	6	9.197606	0.231189
4	7	3.347099	0.635292	8	7	9.02804	0.235531
4	8	3.262009	0.651864	8	8	8.541503	0.248947
4	-1	0.311404	6.82838	8	-1	0.36999	5.747142

Cores	Threads	Time Taken	Speedup target AVX	Cores	Threads	Time Taken	Speedup target AVX
1	1	2.916589	1	5	1	8.099869	0.360079
1	2	1.550587	1.880958	5	2	6.956095	0.419285
1	3	1.048095	2.782753	5	3	6.476902	0.450306
1	4	0.794267	3.672051	5	4	6.254609	0.46631
1	5	0.649557	4.49012	5	5	6.189015	0.471253
1	6	0.548	5.322243	5	6	6.112422	0.477158
1	7	0.481715	6.054595	5	7	6.078189	0.479845
1	8	0.426173	6.843674	5	8	6.054059	0.481758
1	-1	0.211219	13.808365	5	-1	0.254607	11.455258
2	1	3.890887	0.749595	6	1	9.619312	0.303201
2	2	2.982448	0.977918	6	2	8.367636	0.348556
2	3	2.655546	1.098301	6	3	7.913428	0.368562
2	4	2.481232	1.17546	6	4	7.672577	0.380132
2	5	2.402745	1.213857	6	5	7.614653	0.383023
2	6	2.34429	1.244125	6	6	7.630859	0.38221
2	7	2.321025	1.256595	6	7	7.590216	0.384256
2	8	2.272743	1.28329	6	8	7.483798	0.38972
2	-1	0.218913	13.323051	6	-1	0.278638	10.467305
3	1	5.265418	0.553914	7	1	11.211408	0.260145
3	2	4.086385	0.713733	7	2	9.970059	0.292535
3	3	3.715794	0.784917	7	3	9.655147	0.302076
3	4	3.562007	0.818805	7	4	9.408537	0.309994
3	5	3.44378	0.846915	7	5	9.151556	0.318699
3	6	3.370104	0.86543	7	6	9.288197	0.31401
3	7	3.356991	0.86881	7	7	9.237996	0.315717
3	8	3.331537	0.875448	7	8	9.319925	0.312941
3	-1	0.219955	13.259935	7	-1	0.29834	9.776058
4	1	6.670429	0.437242	8	1	12.692486	0.229789
4	2	5.515858	0.528764	8	2	11.375879	0.256384
4	3	4.977613	0.585941	8	3	11.293126	0.258262
4	4	4.758621	0.612906	8	4	10.942658	0.266534

4	5	4.656166	0.626393	8	5	10.833622	0.269216
4	6	4.743893	0.614809	8	6	10.842143	0.269005
4	7	4.549008	0.641148	8	7	10.791316	0.270272
4	8	4.507349	0.647074	8	8	10.681262	0.273057
4	-1	0.220061	13.253548	8	-1	0.299689	9.732052

AVX Intrinsics

These are functions that the compiler replaces with the proper assembly instructions. For our implementation we used Newton's method. We created an array of a __m256 type and made it the length of 1,875,000 since the _mm256_set_ps() function allowed us to create an array of size eight, and 8*1,875,000 = 15,000,000. This was the desired size of the number of elements that we wanted to take the square root. In our code we iterated through each of the 1,875,000 indexes of our array and performed Newton's method for the set of _mm256 elements (8 elements at a time) in parallel.

	Sequential	Parallel
15,000,000 elements	0.686316s	0.481908s
30,000,000 elements	1.369438s	0.966046s

As the above run times indicate, There is a significant speedup when implementing the algorithm by using AVX intrinsics rather than performing the algorithm sequentially. AVX's ability to execute faster is due to the fact that on every iteration of the algorithm, 8 elements are being computed at the same time rather than just 1 element. The reason that there is not a speedup of 8 for the AVX execution could come from the overhead of producing additional storage during execution that the sequential algorithm did not need and an imperfection of implementing the algorithm with AVX. In particular, this implementation will continue to run on every _m256 array until every element that was passed into the algorithm has reached the tolerance threshold. This means that for every array, even if every element within the array has reached the tolerance threshold, if there is even 1 element outside the array that has not reached the threshold, then the algorithm will still be ran across this array too.