

Parallel Square Root Computation

Code Files:

Code files for all the three parts of this problem can be found in their respective folders i.e. Part 1, Part 2 and Part 3.

Performance Comparisons:

In this section, we describe the performance comparisons between serial and parallel methods for finding square root computation using ISPC on a single CPU core (no tasks launched) and multi-core (with tasks). Also, we explain execution of ISPC code with different launch tasks i.e. different threads so as to partition computation accordingly.

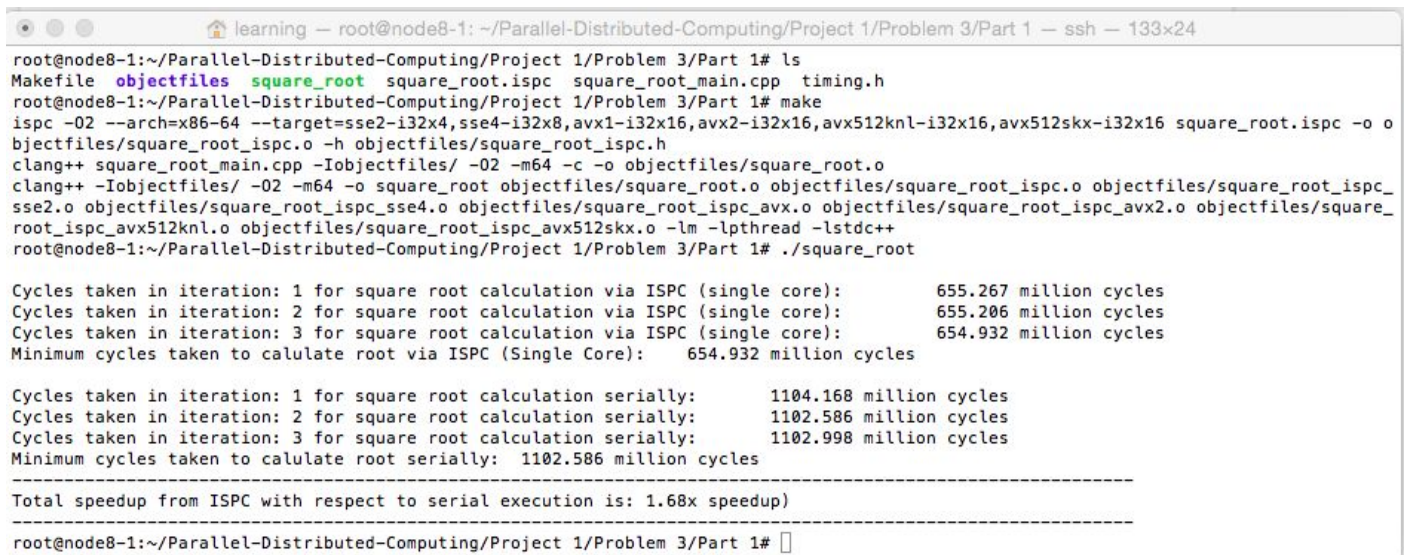
Also, we will see the effects of Intrinsics AVX instructions on the speed up by comparing the speeds between manually written AVX program with AVX instructions, AVX instructions generated from ISPC program and SSE4 instructions generated from ISPC program.

1. Comparison between serial and SIMD parallel program

NOTE: Please refer to the code files present in **Part 1** folder.

Results:

Below is the screenshot of results of executing serial and SIMD parallel execution using ISPC on single core:



```
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1# ls
Makefile  objectfiles  square_root  square_root.ispc  square_root_main.cpp  timing.h
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1# make
ispc -O2 --arch=x86-64 --target=sse2-i32x4,sse4-i32x8,avx1-i32x16,avx2-i32x16,avx512knl-i32x16,avx512skx-i32x16 square_root.ispc -o o
bjectfiles/square_root_ispc.o -h objectfiles/square_root_ispc.h
clang++ square_root_main.cpp -Iobjectfiles/ -O2 -m64 -c -o objectfiles/square_root.o
clang++ -Iobjectfiles/ -O2 -m64 -o square_root objectfiles/square_root.o objectfiles/square_root_ispc.o objectfiles/square_root_ispc_
sse2.o objectfiles/square_root_ispc_sse4.o objectfiles/square_root_ispc_avx.o objectfiles/square_root_ispc_avx2.o objectfiles/square_
root_ispc_avx512knl.o objectfiles/square_root_ispc_avx512skx.o -lm -lpthread -lstdc++
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1# ./square_root

Cycles taken in iteration: 1 for square root calculation via ISPC (single core):      655.267 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC (single core):      655.206 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC (single core):      654.932 million cycles
Minimum cycles taken to calculate root via ISPC (Single Core):      654.932 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1104.168 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1102.586 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1102.998 million cycles
Minimum cycles taken to calculate root serially: 1102.586 million cycles

-----
Total speedup from ISPC with respect to serial execution is: 1.68x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1#
```

As we can see in the screenshot above, we observed a **speedup of 1.68x** when computation is performed by SIMD parallelization that uses ISPC on a single CPU core compared to that of serial execution for finding the square roots of 20 million numbers.

Instructions to execute the code:

1. Move inside the Part 1 folder and you will see a bunch of files which do performance comparison between serial and SIMD parallel execution using ISPC on a single CPU core.
 - a. **square_root_main.cpp**: This is the main file which generates random numbers and executes the functions that calculates square roots of the numbers generated using Newton's method. It calls serial square root function and ISPC square root function on single core.
 - b. **square_root.ispc**: This is the file which contains ispc function that calculates square root of 20 million numbers.
 - c. **Makefile**: This file contains all the instructions that are required to generate **square_root.ispc.h** (header file consisting of ISPC function) and other object files of different target instructions set. Basically, it will compile all the code files and generate an executable file as well i.e. **square_root**
2. Now, execute make file using below command:
make
3. After compiling the code files, run the executable file i.e. **square_root** to get the results:
./square_root
4. In each of the methods (serial and parallel), we are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison.

2. Comparison between serial, ISPC on single core and multi-core (using tasks) program

NOTE: Please refer to the code files present in **Part 2** folder.

Results:

Below is the screenshot of results of executing serial, ISPC on single core (with no tasks) and ISPC on multi-core (using tasks)

Serial v/s ISPC (single core i.e. 1 thread/task):

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1 — ssh — 133x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1# ls
Makefile  objectfiles  square_root  square_root.ispc  square_root_main.cpp  timing.h
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1# make
ispc -O2 --arch=x86-64 --target=sse2-i32x4,sse4-i32x8,avx1-i32x16,avx2-i32x16,avx512knl-i32x16,avx512skx-i32x16 square_root.ispc -o o
bjectfiles/square_root.ispc.o -h objectfiles/square_root.ispc.h
clang++ square_root_main.cpp -Iobjectfiles/ -O2 -m64 -c -o objectfiles/square_root.o
clang++ -Iobjectfiles/ -O2 -m64 -o square_root objectfiles/square_root.o objectfiles/square_root.ispc.o objectfiles/square_root.ispc_
sse2.o objectfiles/square_root.ispc_sse4.o objectfiles/square_root.ispc_avx.o objectfiles/square_root.ispc_avx2.o objectfiles/square_
root.ispc_avx512knl.o objectfiles/square_root.ispc_avx512skx.o -lm -lpthread -lstdc++
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1# ./square_root

Cycles taken in iteration: 1 for square root calculation via ISPC (single core):      655.267 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC (single core):      655.206 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC (single core):      654.932 million cycles
Minimum cycles taken to calculate root via ISPC (Single Core):    654.932 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1104.168 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1102.586 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1102.998 million cycles
Minimum cycles taken to calculate root serially:    1102.586 million cycles

-----
Total speedup from ISPC with respect to serial execution is: 1.68x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 1#
```

Speedup by ISPC (1-thread) i.e. single core: **1.68x**

Serial v/s ISPC (multicores - 2 threads/tasks):

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2 — ssh — 133x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2# ./square_root_tasks 2

Cycles taken in iteration: 1 for square root calculation via ISPC with 2 threads:      368.416 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC with 2 threads:      368.669 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC with 2 threads:      368.646 million cycles
Minimum cycles taken to calculate root via ISPC with 2 threads:  368.416 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1100.177 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1099.848 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1099.910 million cycles
Minimum cycles taken to calculate root serially:  1099.848 million cycles

-----
Total speedup from ISPC with 2 threads compared to serial execution is: 2.99x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2#
```

Speedup by ISPC (2-threads): 2.99x

Serial v/s ISPC (multicores - 3 threads/tasks):

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2 — ssh — 133x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2# ./square_root_tasks 3

Cycles taken in iteration: 1 for square root calculation via ISPC with 3 threads:      225.359 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC with 3 threads:      221.163 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC with 3 threads:      221.160 million cycles
Minimum cycles taken to calculate root via ISPC with 3 threads:  221.160 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1100.238 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1100.008 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1100.054 million cycles
Minimum cycles taken to calculate root serially:  1100.008 million cycles

-----
Total speedup from ISPC with 3 threads compared to serial execution is: 4.97x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2#
```

Speedup by ISPC (3-threads): 4.97x

Serial v/s ISPC (multicores - 4 threads/tasks):

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2 — ssh — 133x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2# ./square_root_tasks 4

Cycles taken in iteration: 1 for square root calculation via ISPC with 4 threads:      181.437 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC with 4 threads:      176.719 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC with 4 threads:      164.824 million cycles
Minimum cycles taken to calculate root via ISPC with 4 threads:  164.824 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1100.208 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1099.892 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1099.899 million cycles
Minimum cycles taken to calculate root serially:  1099.892 million cycles

-----
Total speedup from ISPC with 4 threads compared to serial execution is: 6.67x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2#
```

Speedup by ISPC (4-threads) : 6.67x

Serial v/s ISPC (multicores - 5 threads/tasks):

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2 — ssh — 127x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2# ./square_root_tasks 5

Cycles taken in iteration: 1 for square root calculation via ISPC with 5 threads:      261.284 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC with 5 threads:      260.945 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC with 5 threads:      260.825 million cycles
Minimum cycles taken to calculate root via ISPC with 5 threads:  260.825 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1100.245 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1100.072 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1099.880 million cycles
Minimum cycles taken to calculate root serially:  1099.880 million cycles

-----
Total speedup from ISPC with 5 threads compared to serial execution is: 4.22x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2#
```

Speedup by ISPC (5-threads) : 4.22x

Serial v/s ISPC (multicores - 6 threads/tasks):

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2 — ssh — 127x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2# ./square_root_tasks 6

Cycles taken in iteration: 1 for square root calculation via ISPC with 6 threads:      220.133 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC with 6 threads:      220.190 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC with 6 threads:      220.404 million cycles
Minimum cycles taken to calculate root via ISPC with 6 threads:  220.133 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1100.238 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1100.016 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1099.781 million cycles
Minimum cycles taken to calculate root serially:  1099.781 million cycles

-----
Total speedup from ISPC with 6 threads compared to serial execution is: 5.00x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2#
```

Speedup by ISPC (6-threads) : 5.00x

Serial v/s ISPC (multicores - 7 threads/tasks):

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2 — ssh — 127x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2# ./square_root_tasks 7

Cycles taken in iteration: 1 for square root calculation via ISPC with 7 threads:      201.919 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC with 7 threads:      189.215 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC with 7 threads:      189.430 million cycles
Minimum cycles taken to calculate root via ISPC with 7 threads:  189.215 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1100.189 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1099.809 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1099.950 million cycles
Minimum cycles taken to calculate root serially:  1099.809 million cycles

-----
Total speedup from ISPC with 7 threads compared to serial execution is: 5.81x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2#
```

Speedup by ISPC (7-threads) : 5.81x

Serial v/s ISPC (multicores - 8 threads/tasks):

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2 — ssh — 127x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2# ./square_root_tasks 8

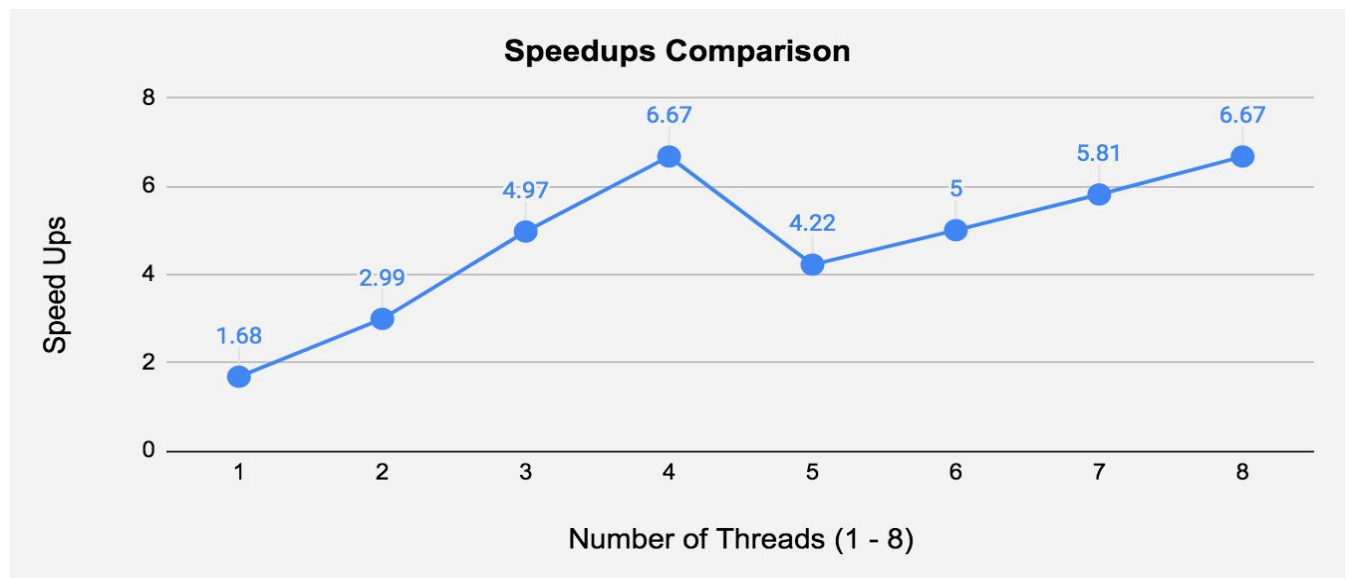
Cycles taken in iteration: 1 for square root calculation via ISPC with 8 threads:      181.293 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC with 8 threads:      176.614 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC with 8 threads:      164.765 million cycles
Minimum cycles taken to calculate root via ISPC with 8 threads:  164.765 million cycles

Cycles taken in iteration: 1 for square root calculation serially:      1100.284 million cycles
Cycles taken in iteration: 2 for square root calculation serially:      1099.803 million cycles
Cycles taken in iteration: 3 for square root calculation serially:      1099.887 million cycles
Minimum cycles taken to calculate root serially:  1099.803 million cycles

-----
Total speedup from ISPC with 8 threads compared to serial execution is: 6.67x speedup)
-----
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2#
```

Speedup by ISPC (7-threads) : 6.67x

Graph for speedups comparison:



Discussion:

In this section, we went through the effects of threads while performing parallel execution of ISPC program that computes square root of 20 million numbers and the speed up with respect to sequential execution method for performing square root.

1. When the number of threads/tasks is 1, the speedup is the same as that of Part 1 of this problem i.e. SIMD parallelization using ISPC execution which is very much valid as it runs computation on the same single core with only 1 thread/worker/task doing the needful.

2. As we increase the number of threads i.e. launch more tasks the speed up increases linearly but not consistently from 1 to 8 threads. It reaches the peak speed up for 4 threads as seen in the above graph thus achieving speed up of **6.67x** with respect to sequential execution.
3. When we increase threads before 4, we see a linear drop in speed up for 5 threads and again as well increase the threads it increases and reaches the same peak at 8 threads i.e. same as 4 threads thus achieving speed up of **6.67x**.
4. Thus, the performance decreases at some point and one of the reasons might be because of either thread inactivity resulting in under utilization of hardware.
5. One more major reason which we think that achieved the highest speedup at 4 and 8 threads is the machine has 4 cores and each core executes 1 thread. Thus, at a given point in time all threads would be running in case of 4 threads thus utilizing 4 cores fully and also for 8 threads, 4 threads in one clock cycle and other 4 in other clock cycles would be running in 4 cores thus fully utilizing it.

Instructions to execute the code:

1. Move inside the Part 2 folder and you will see a bunch of files which do performance comparison between serial and ISPC on single and multicores by using threads/workers/tasks for computing the square root of 20 million numbers.
 - a. **square_root_main.cpp**: This is the main file which generates random numbers and executes the functions that calculates square roots of the numbers generated using Newton's method. It calls serial square root function and also launches tasks for computation by calling ISPC square root function based on the user inputted number of threads.
 - b. **square_root_tasks.ispc**: This is the file which contains ispc function that calculates square root of 20 million numbers and accepts a number of threads as primary parameter which is used to launch the corresponding number of tasks.
 - c. **Makefile**: This file contains all the instructions that are required to generate **square_root_tasks.ispc.h** (header file consisting of ISPC function) and other object files of different target instructions set. Basically, it will compile all the code files and generate an executable file as well i.e. **square_root_tasks**
2. Now, execute make file using below command:
3. After compiling the code files, run the executable file i.e. **square_root_tasks** to get the results:

./square_root_tasks 2

NOTE: If you don't pass the number of threads as an argument while running the executable file, you will get an error message and you won't be able to proceed further. Check the below error message:

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2 — ssh
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2# ./square_root_tasks

ERROR: You must pass number of threads for executing ISPC code with launch tasks in the arguments...!!
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 2#
```

4. In each of the methods (serial and parallel), we are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison

3. Effects of AVX Intrinsics (manual and from ISPC) on speedup

In this section we will look at the effect of AVX Intrinsics on performance and compare the performance of programs written manually in AVX with that of AVX instruction set generated from ISPC on different targets and also with SSE4 target.

Results:

Manual AVX Program:

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ManualAVX — ssh — 125x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ManualAVX# make
gcc -mavx -o square_root_avx square_root_avx.cpp -march=native -O3 -lstdc++ -lm --std=c99
cc1plus: warning: command line option '-std=c99' is valid for C/ObjC but not for C++ [enabled by default]
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ManualAVX# ./square_root_avx

Cycles taken in iteration: 1 for square root calculation via AVX Intrinsics:      0.009 million cycles
Cycles taken in iteration: 2 for square root calculation via AVX Intrinsics:      0.000 million cycles
Cycles taken in iteration: 3 for square root calculation via AVX Intrinsics:      0.000 million cycles
Minimum cycles taken to calculate root via Manual AVX Intrinsics      0.000254 million cycles

root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ManualAVX#
```

As seen in the above screenshot, we can see that it takes very less machine cycles to complete the entire execution of computing the square root of 20 million numbers compared to that of earlier programs. In this program, we have used the inbuilt AVX Intrinsics library that loads the data in the vectors and computes the square root via the AVX square root function. And that might be one of the reasons why we can see the best performance until now.

AVX generated by ISPC: (avx2-i32x8)

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_AVX2 — ssh — 125x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_AVX2# make
ispc -O2 --arch=x86-64 --target=avx2-i32x8 square_root.ispc -o objectfiles/square_root_ispc.o -h objectfiles/square_root_ispc
.h
clang++ square_root_main.cpp -Iobjectfiles/ -O2 -m64 -c -o objectfiles/square_root.o
clang++ -Iobjectfiles/ -O2 -m64 -o square_root_avx2 objectfiles/square_root.o objectfiles/square_root_ispc.o -lm -lpthread -l
stdc++
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_AVX2# ./square_root_avx2

Cycles taken in iteration: 1 for square root calculation via ISPC (single core):      870.538 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC (single core):      869.640 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC (single core):      868.764 million cycles
Minimum cycles taken to calculate root via ISPC (Single Core) with target as avx2 :      868.764 million cycles

root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_AVX2#
```

As seen in the above screenshot, we can see that the performance in terms of minimum machine cycles taken to calculate the square root of numbers is low. One of the major reasons why the performance is lower for AVX instructions generated by ISPC compilers with that manually written AVX program by using inbuilt functions is because AVX code is very faster than that of one generated by ISPC compiler.

AVX generated by ISPC: (avx2-i32x16)

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_AVX2 — ssh — 125x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_AVX2# make
ispc -O2 --arch=x86-64 --target=avx2-i32x16 square_root.ispc -o objectfiles/square_root_ispc.o -h objectfiles/square_root_ispc.h
clang++ square_root_main.cpp -Iobjectfiles/ -O2 -m64 -c -o objectfiles/square_root.o
clang++ -Iobjectfiles/ -O2 -m64 -o square_root_avx2 objectfiles/square_root.o objectfiles/square_root_ispc.o -lm -lpthread -lstdc++
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_AVX2# ./square_root_avx2

Cycles taken in iteration: 1 for square root calculation via ISPC (single core):      654.569 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC (single core):      654.347 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC (single core):      654.667 million cycles
Minimum cycles taken to calculate root via ISPC (Single Core) with target as avx2 :    654.347 million cycles
```

Now, as seen in above image the AVX instruction set program generated by ISPC compiler for target as avx2-i32x16 performs better than the one generated for target as avx2-i32x8. This is very obvious as we have moved to a higher configuration of target where we are asking the compiler to divide the computation among 16 gang sizes or i.e. 16 instances in a gang compared to that of 8 in earlier cases with 32-bit masking.

However, the performance is still lower than that of AVX code written manually by using the AVX instructions and inbuilt functions due to the same reasons that were mentioned earlier.

SSE4 generated by ISPC: (sse4-i32x8)

```
learning — root@node8-1: ~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_SSE4 — ssh — 125x24
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_SSE4# make
ispc -O2 --arch=x86-64 --target=sse4-i32x8 square_root.ispc -o objectfiles/square_root_ispc.o -h objectfiles/square_root_ispc.h
clang++ square_root_main.cpp -Iobjectfiles/ -O2 -m64 -c -o objectfiles/square_root.o
clang++ -Iobjectfiles/ -O2 -m64 -o square_root_sse4 objectfiles/square_root.o objectfiles/square_root_ispc.o -lm -lpthread -lstdc++
root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_SSE4# ./square_root_sse4

Cycles taken in iteration: 1 for square root calculation via ISPC (single core):      970.035 million cycles
Cycles taken in iteration: 2 for square root calculation via ISPC (single core):      972.547 million cycles
Cycles taken in iteration: 3 for square root calculation via ISPC (single core):      972.374 million cycles
Minimum cycles taken to calculate root via ISPC (Single Core) with target as sse4 :    970.035 million cycles

root@node8-1:~/Parallel-Distributed-Computing/Project 1/Problem 3/Part 3/ISPC_SSE4#
```

As seen in the above image, the performance of the program for the instruction set target as SSE4 is lowest compared to that of AVX code written manually or generated by ISPC compiler. One of the reasons is because gang size is only 8 but that is not of much importance because even with size 8 the performance is lower than avx2-i32x8 target instruction set. The only primary reason is AVX code or program is very fast as it deals with vectors that speeds up the program execution because all the vectors work in parallel for performing the computation.

Discussion:

1. Majoring of the points is already discussed in each topic of performance comparison between manually written AVX, ISPC generated AVX and SSE4 instruction sets.

2. Manually written AVX programs work the fastest among every other program followed by AVX programs generated by ISPC compilers.
3. Some reasons are AVX programs are designed to be faster than ISPC programs and AVX programs deal with vectors that help in speedup during execution.
4. Irrespective of this, ISPC is widely used because of the ease of programming and also understanding the program is very easy.

Instructions to execute the code:

Move inside the Part 3 folder and you will see 3 other folders - ISPC_AVX2, ISPC_SSE4 and ManualAVX

ManualAVX:

This folder contains the manually written AVX code that we had discussed in the very beginning of this topic.

1. Move inside the ManualAVX folder and you will see a bunch of files which will compute square root of 20 million numbers via Manually written AVX program with the help of inbuilt libraries.
 - a. ***square_root_avx.cpp***: This is the main file which generates random numbers and executes the functions that calculates square roots of the numbers generated using Newton's method by using vectors provided by AVX Intrinsics.
 - b. **Makefile**: This file contains all the instructions that are required to generate object files of the avx program and also executable files. Basically, it will compile all the AVX Intrinsics code and generate an executable file as well i.e. ***square_root_avx***
2. Now, execute make file using below command:
make
3. After compiling the code files, run the executable file i.e. ***square_root_avx*** to get the results:
./square_root_avx
4. We are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison.

ISPC_AVX2:

This folder contains the ISPC program that can be compiled to generate ***avx2-i32x-8*** and ***avx2-i32x-16*** target instructions sets for AVX.

1. Move inside the ISPC_AVX2 folder and you will see a bunch of files which will compute the square root of 20 million numbers via ISPC program that can be compiled in different AVX target instructions sets using **Makefile**.
 - a. ***square_root_main.cpp***: This is the main file which generates random numbers and executes the functions that calculate square roots of the numbers generated using Newton's method by SIMD based ISPC program.
 - b. **Makefile**: This file contains all the instructions that are required to generate object files of the ISPC program and also executable files. Basically, it will compile all the ISPC code and generate an executable file as well i.e. ***square_root_avx2***

- c. This executable file will have a target instruction sets of AVX2 i.e. **avx2-i32x-8** and **avx2-i32x-16**
 - d. For generating **avx2-i32x-8** instructions set target related executable file, make sure the **target** is assigned as **avx2-i32x-8** and similarly for **avx2-i32x-16** the **target** is assigned as **avx2-i32x-16** in the **Makefile**. Thus, you will need to change the **target** variable in Makefile for each case manually.
2. Now, execute make file using below command:
make
 3. After compiling the code files, run the executable file i.e. square_root_avx2 to get the results:
./square_root_avx2
 4. We are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison.

ISPC_SSE4:

This folder contains the ISPC program that can be compiled to generate **avx2-i32x-8** target instruction set for SSE4.

1. Move inside the ISPC_SSE4 folder and you will see a bunch of files which will compute the square root of 20 million numbers via the ISPC program that can be compiled in the SSE4 target instruction set using **Makefile**.
 - a. square_root_main.cpp: This is the main file which generates random numbers and executes the functions that calculate square roots of the numbers generated using Newton's method by SIMD based ISPC program.
 - b. Makefile: This file contains all the instructions that are required to generate object files of the ISPC program and also executable files. Basically, it will compile all the ISPC code and generate an executable file as well i.e. square_root_sse4
 - c. This executable file will have a target instruction set of SSE4 i.e. **sse4-i32x-8**
2. Now, execute make file using below command:
make
3. After compiling the code files, run the executable file i.e. square_root_sse4 to get the results:
./square_root_sse4
4. We are calculating the square roots 3 times and choose the lowest time among those iterations for better comparison.

Machine Specifications:

We won't be 100% sure about this as we had used **node8-1** from the **grid machine** in the Orbit Systems for doing Problem 3.

CPU: 4 Core and 1 thread per core

Memory: 8 GB

OS: Ubuntu

Image: mariasfirstimage.ndz