15-418/15-618 Parallel Computer Architecture and Programming

Homework 1

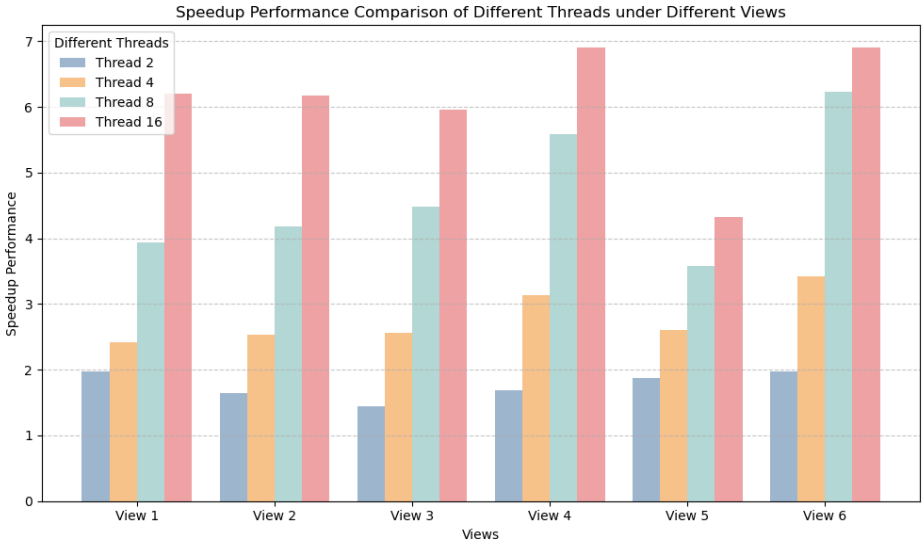
Siqi Guo(AndrewID: siqiguo) December 29, 2024

1. **Problem 1: Parallel Fractal Generation Using Pthreads (15 points)**

   i. This type of problem decomposition is referred as Spatial Decomposition since different spatial regions of the image are computed by different processors.

   Note that the processor (eight 3.2 GHz Intel Core i7 processors) only has 8 cores, but each core supports two hyper-threads.

   ii. We partition the image generation work into the appropriate number of horizontal blocks.



   - Speedup is not linear in the number of cores used.
   - The workload is not necessarily evenly distributed among the threads.
   - My measurements show that the elapsed time is not same for each thread, which explains the non-linear speedup due to non-evenly workload.

   iii. Modify the mapping of work to threads to improve speedup to almost 8x on the first two views of the Mandelbrot set.

   To decompose the work into row-wise blocks, we can assign each thread to compute a row of the image. This way, the workload is more evenly distributed among the threads, we could reach speedup around 7.40x ~7.53x. For view 1, the speedup from 4 to 8 threads is from 3.81x to 7.47x, but from 8 to 16 threads, the speedup is from 7.47x to 7.40x. For view 2, the speedup from 4 to 8 threads is from 3.72x to 7.29x, but from 8 to 16 threads, the speedup is from 7.29x to 7.28x.

   In my decomposition policy, the scaling behavior are different from 4 to 8 threads and 8 to 16 threads, which indicates 4 to 8 threads speedup is almost linear because all threads have access to independent physical cores. However, 8 to 16 threads speedup is not linear because the threads are sharing the same physical core due to hyper-threading.

2. Problem 2: Vectorizing Code Using SIMD Intrinsics (20 points)

i. The implementation of `clampedExpVector()` should work with any combination of input array size N and vector width W.

ii. Run `./vrun -s 10000` and sweep the vector width over the values 2, 4, 8, 16, 32. Record the resulting vector utilization.

VECTOR_WIDTH W	2	4	8	16	32
Total Vector Instructions	276613	141698	71238	35628	17787
Vector Utilization	89.066132 %	88.370866 %	88.216787 %	88.211344 %	88.212072 %

Table 1: Vector Utilization for Different Vector Widths

The vector utilization decreases as W increases a bit, but the difference is not significant. The degree of sensitivity the utilization has on the vector width is not very high.
The higher W is, the more vector instructions are executed.

iii. `arraySumVector()` has been implemented, and the results passed the correctness test as follows.

```
siqiguo@ghc28:~/private/15-618-Parallel-Computing/asst1/prog2_vecintrin$ make
g++ -I../common logger.o CMU418intrin.o main.cpp functions.cpp -o vrun
siqiguo@ghc28:~/private/15-618-Parallel-Computing/asst1/prog2_vecintrin$ ./vrun -s 8
CLAMPED EXPONENT (required)
Results matched with answer!
****************** Printing Vector Unit Statistics ******************
Vector Width:              8
Total Vector Instructions: 60
Vector Utilization:        90.000000%
Utilized Vector Lanes:     432
Total Vector Lanes:        480
*********************** Result Verification ************************
Passed!!!

ARRAY SUM (bonus)
Passed!!!
```

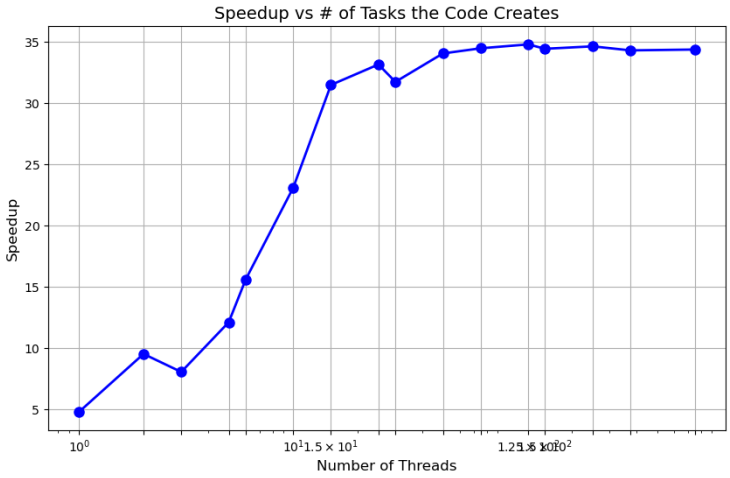
3. Problem 3: Parallel Fractal Generation Using ISPC (15 points)

i. A Few ISPC Basics (7 of 15 points)

- maximum speedup: 8x, as 8-wide AVX2 instruction operates on 8 data elements simultaneously. SIMD parallelism.
- The reason why the number I observe be less than this ideal: workload imbalance across lanes, branch divergence.

ii. ISPC Tasks (8 of 15 points)

- The speedup from task ISPC (the mandelbrot multicore ispc) is 9.54x, compared to from ISPC (the mandelbrot ispc) 4.83x. The speedup over the version that does not partition that computation into tasks is $9.54/4.83 = 1.975$x.
- The speedup could reach 34.81x when we set the number of tasks to 125.



- ISPC's task abstraction efficiently utilizes SIMD hardware to process data in parallel by mapping program instances to SIMD lanes, minimizing overhead by working directly with vector instructions. This is well-suited for fine-grained, data-parallel tasks where computations align with SIMD capabilities.
  In contrast, Pthreads provide a general-purpose threading abstraction that relies on the operating system for scheduling and synchronization, which introduces higher overheads due to thread creation, management, and context switching.
  While ISPC excels in data-level parallelism, Pthreads are more flexible for coarse-grained parallelism across multiple cores.

4. **Problem 4: Iterative Square Root (10 points)**

   i. The sqrt program has been built and run. The speedup due to SIMD (no tasks) parallelization is 4.77x. The speedup due to multi-core parallelization is 34.69x.

   ii. The speedup under different scenarios is as follows.

Case	Random	initGood	initBad
SIMD Speedup	4.77x	6.88x	0.95x
Multi-Core Speedup	34.68x	50.16x	6.89x

Table 2: Speedup under Different Scenarios

The `initGood()` change the value to 2.999. The initialized value of 2.999 could result in the maximum iteration, and accelerate the paralleled computation to the largest extent. This modification improves both SIMD and multi-core speedup.

The `initBad()` change the value to 1.0. The value around 1.0 results in the minimum iteration and the least absolute computation time. There is not much time to decrease with parallelism. This modification decreases SIMD and multi-core speedup.

5. **Problem 5: BLAS saxpy (10 points)**

The saxpy routine in the BLAS (Basic Linear Algebra Subproblems) library that is widely used (and heavily optimized) on many systems.

The **saxpy** function computes the operation: $\mathbf{r} = a\mathbf{x} + \mathbf{y}$ where a is a scalar, and $\mathbf{r}, \mathbf{x}, \mathbf{y}$ are vectors of size N containing single-precision floating-point values. The term **saxpy** is an acronym for **"single-precision a x plus y"**.

i. The speedup due to SIMD parallelization is 1x. The speedup from ISPC with tasks is 1.03x.

Table 3: SAXPY Performance Comparison

Method	Time (ms)	Bandwidth (GB/s)	GFLOPS
SAXPY Serial	11.238	26.520	1.780
SAXPY Streaming	11.238	26.520	1.780
SAXPY ISPC	11.246	26.500	1.778
SAXPY Task ISPC	10.920	27.291	1.831
(1.00x speedup from streaming)			
(1.00x speedup from ISPC)			
(1.03x speedup from task ISPC)			

I do not think we could achieve linear speedup as the memory bandwidth could be the bottleneck.

ii. One value is loaded from vector x `(N x sizeof(float))`. One value is loaded from vector y `(N x sizeof(float))`. One value is written to the result vector r `(N x sizeof(float))`. However, result should be stored in the cache, thus a cache miss will lead to 2 N (read from memory to cache, then back to memory).

iii. In addition to the memory bandwidth, the performance of SAXPY might be also limited by poor cache utilization, high cache misses, low pre-fetching efficiency, and the latency of FP operations or DRAM.

iv. Improve the performance of saxpy by reducing the memory requirement (modify `saxpyStreaming()`).

Table 4: SAXPY Performance Comparison After Streaming

Method	Time (ms)	Bandwidth (GB/s)	GFLOPS
SAXPY Serial	11.280	26.421	1.773
SAXPY Streaming	8.209	36.302	2.436
SAXPY ISPC	11.122	26.797	1.798
SAXPY Task ISPC	10.900	27.342	1.835
(1.37x speedup from streaming)			
(1.01x speedup from ISPC)			
(1.03x speedup from task ISPC)			

Overall, the cache does matter in this problem.