CS 6210: Scientific and Data Computing I Fall 2024

Assignment 2

Milena Belianovich
November 17, 2024

For code development and testing for this assignment, use any computer with OpenMP, e.g., CADE Lab or your laptop. After code development, performance is to be reported on a 20-core node on the Lonepeak CHPC cluster (using batch submission). Note that you should not perform direct execution of programs on CHPC login nodes (account will get suspended if some threshold of total interactive CPU time is exceeded.

Submission: 1) A PDF report providing explanations as specified for each question, 2) Submission of the opt.c files for questions 2, 3, 4, and 5 (optional, extra credit question).

For this assignment, use the clang C compiler with the following options to get feedback on which loops were vectorized and which could not be vectorized:

clang -O3 -fopenmp -Rpass-missed=loop-vectorize -Rpass=loop-vectorize

1 Problems

1. (25 points)Compile and execute the program vec1main.c. It invokes functions vec1a, vec1b, and vec1c, which execute loops enclosing the following 3 statements: vec1a: w[i] = w[i]+1; vec1b: w[i] = w[i+1]+1; vec1c: w[i] = w[i-1]+1;

   Which of these would you expect to vectorize and why? Do your expectations match the vectorization report from clang? Is there a difference in achieved performance for the statements that are vectorized?

   Solution.

   Which of these would you expect to vectorize and why?

   Expected to Vectorize:

   - w[i] = w[i] + 1 (in vec1a.c) and w[i] = w[i+1] + 1 (in vec1b.c).
   - These loops are expected to vectorize because they do not have dependencies that prevent parallel execution. In w[i] = w[i] + 1, each loop iteration operates independently on different elements of w, allowing parallelism without any issue. Similarly, in w[i] = w[i+1] + 1, while each iteration reads from w[i+1], it does not modify it, allowing multiple iterations to be computed simultaneously.

   Not Expected to Vectorize:

- w[i] = w[i-1] + 1 (in vec1c.c) was not expected to vectorize because of the backward dependency on w[i-1]. This dependency forces each iteration to wait for the previous one to complete, making parallel execution infeasible in this case.

Do your expectations match the vectorization report from clang?

Yes, the expectations match the clang vectorization report. Both w[i] = w[i] + 1 and w[i] = w[i+1] + 1 were successfully vectorized with a vectorization width of 4 and an interleaved count of 2, confirming that clang could handle these loops in parallel. Conversely, w[i] = w[i-1] + 1 was not vectorized, as anticipated, due to the loop-carried dependency.

Is there a difference in achieved performance for the statements that are vectorized?

Yes, there is a significant difference in performance between the vectorized and non-vectorized statements:

- CADE
  - For w[i] = w[i] + 1 and w[i] = w[i+1] + 1, the performance was high, with GFLOPS values ranging from approximately 13.13 to 13.46.
  - For w[i] = w[i-1] + 1, which was not vectorized, the performance was much lower, with GFLOPS values between 1.17 and 1.18.
- Lonepeak CHPC:
  - w[i] = w[i] + 1:
    - Performance (GFLOPS): Min: 6.17; Max: 6.36
    - This loop was vectorized successfully with no dependencies.
  - w[i] = w[i+1] + 1:
    - Performance (GFLOPS): Min: 6.10; Max: 6.24
    - This loop was also vectorized as it only reads from w[i+1].
  - w[i] = w[i-1] + 1:
    - Performance (GFLOPS): Min: 0.88; Max: 0.88
    - This loop was not vectorized due to a backward dependency on w[i-1].

This performance difference highlights the effectiveness of vectorization: the vectorized loops achieve substantially higher throughput by executing multiple operations simultaneously, whereas the non-vectorized loop remains constrained by its sequential dependencies.

2. (25 points) The following loop code in vec2ref.c is not vectorized by clang. Can it be transformed into functionally equivalent code (i.e., satisfies all data dependences) that can be vectorized by clang? Explain why or why not? If modification to enable vectorization is feasible, modify the code in vec2opt.c (initially identical to the code in vec2ref.c) to achieve higher (single thread) performance via vectorization.

```
for (j=0; j<n; j++)
    for(i=1; i<n; i++)
```

```
// A[i][j] = A[i-1][j]+1;
A[i*n+j] = A[(i-1)*n+j]+1;
```

Solution.

Can This Code Be Vectorized?

The primary challenge to vectorization here is the dependency on previous values in the i loop:

- Each iteration of A[i * n + j] = A[(i - 1) * n + j] + 1 depends on the value calculated in the previous row (i.e., A[(i - 1) * n + j]).
- This dependency prevents the i loop from being vectorized directly, as each computation depends on the result from the previous iteration.

To achieve vectorization, we need to restructure the loop to remove the dependency. In this case, one feasible approach is to parallelize the outer j loop instead, as each column (j) can be computed independently of others. Here's how to modify the code to potentially achieve vectorization:

Swap the Order of Loops: Switch the i and j loops so that the j loop becomes the inner loop. This allows vectorization along the j dimension, which does not have dependencies:

```
for (i=1; i<n; i++) {
    for (j=0; j<n; j++)
        {A[i*n+j] = A[(i-1)*n+j]+1;}}
```

Explanation of Why the Transformation Works:

- By making j the inner loop, the compiler can vectorize along the columns for each row (i), where each column calculation is independent. This removes the dependency in the inner loop and allows vectorized execution.

Performance Impact:

- CADE:The optimized version achieves significantly higher performance (GFLOPS values up to 20.55) compared to the reference version (up to 0.71 GFLOPS), demonstrating the effectiveness of vectorization in accelerating computation.
- Lonepeak CHPC:Reference Code: The optimized version achieves higher performance (GFLOPS values up to 9.80) compared to the reference version (up to 0.41 GFLOPS), demonstrating the effectiveness of vectorization in accelerating computation.

3. (25 points) The following loop code in vec3ref.c is not vectorized by clang. Can it be transformed into functionally equivalent code (i.e., satisfies all data dependences) that can be vectorized by clang? Explain why or why not? If feasible, modify the code in vec3opt.c (initially identical to the code in vec3ref.c) to achieve higher (single thread) performance via vectorization.

```
for(i=1; i<n; i++)
    { w[i] = y[i]+1;
    y[i+1] = 2*x[i];}
```

Solution.

Can This Code Be Vectorized?

The main issue here is the dependency between consecutive iterations in the i loop:

- w[i] = y[i] + 1 calculates w[i] based on the current y[i], which does not create any dependency.
- However, y[i + 1] = 2 * x[i] depends on y[i + 1], which may be updated in the next iteration. This causes a dependency that prevents vectorization, as each iteration modifies the y array in a way that subsequent iterations rely on.

To enable vectorization, we need to remove or reframe this dependency. Since each w[i] calculation is independent, but the y[i+1] update prevents direct vectorization, here's a solution:

Separate the Computation of w and y:

- Perform the computation of w[i] = y[i] + 1 for each i independently in a separate loop.
- Afterward, calculate y[i + 1] = 2 * x[i] for each i, again using a separate loop.

This will allow each loop to run independently and eliminate interdependencies, enabling vectorization:

```
for (rep = 0; rep < Reps; rep++) {
    // First, update w[i] independently
    for (i = 0; i < n - 1; i++)
        { w[i] = y[i] + 1; }

    // Then, update y[i + 1] independently
    for (i = 0; i < n - 1; i++)
        { y[i + 1] = 2 * x[i]; }
}
```

Explanation of Why the Transformation Works:

- By separating the computations into two loops, we eliminate interdependencies between iterations in each loop, enabling clang to vectorize both loops independently.

Performance Impact:

- CADE: The optimized code (vec3opt.c) achieved significantly higher performance, with GFLOPS values up to 15.89, compared to the reference code's maximum of 3.59 GFLOPS, showcasing the advantages of vectorization.

- Lonepeak CHPC: The optimized code achieved higher performance, with GFLOPS values up to 5.11, compared to the reference code's maximum of 1.88 GFLOPS, showcasing the advantages of vectorization.

4. (25 points) The following loop code in vec4ref.c is not vectorized by clang. Can it be transformed into functionally equivalent code (i.e., satisfies all data dependences) that can be vectorized by clang? Explain why or why not? If feasible, modify the code in vec4opt.c (initially identical to the code in vec4ref.c) to achieve higher (single thread) performance via vectorization.

```
for(i=1; i<n; i++)
    { w[i+1] = y[i]+1;
    y[i+1] = x[i]+w[i];}
```

Solution.

Can This Code Be Vectorized?

The main obstacle here is the dependency between consecutive iterations in the i loop:

- w[i + 1] = y[i] + 1 does not have a dependency within the same iteration, but each w[i + 1] relies on y[i] from the previous iteration.
- y[i + 1] = x[i] + w[i] depends on the w[i] computed in the same iteration, further complicating vectorization.

These dependencies prevent the compiler from vectorizing the i loop directly because each iteration requires the results of the previous one.

To enable vectorization, we explored several techniques to eliminate or mitigate these dependencies. Here is a summary of each method tried, represented in pseudocode, along with an explanation of why it did not work.

Methods Explored for Optimization.

(a) Loop Unrolling:

- Description: This approach attempted to reduce loop overhead by processing two elements per iteration, computing w[i+1] and y[i+1] in one part of the iteration and w[i+2] and y[i+2] in the other.
- Pseudocode:

```
for  (i=0; i<n-2; i+=2)
  { w[i+1]=y[i]+1;
    y[i+1]=x[i]+w[i];
    w[i+2]=y[i+1]+1;
    y[i+2]=x[i+1]+w[i+1]; }
if  (i<n-1) {
  { w[i+1]=y[i]+1;
    y[i+1]=x[i]+w[i];  }
```

- Outcome: This method resulted in incorrect results due to the dependency between w[i+1] and y[i+1]. Each value calculated for y[i+1] depends on w[i+1], so separating these into two different parts of a loop caused incorrect data flow and failed to improve performance.
- Performance: The GFLOPS values were lower than the original loop, as this structure introduced complexity without eliminating dependencies.

(b) Loop Splitting:

- Description: This approach separated the calculations for w[i+1] and y[i+1] into two different loops, hoping to avoid cross-iteration dependencies and allow vectorization.
- Pseudocode:

```
// First loop to update w
for (i=0; i<n-1; i++)
    { w[i+1]=y[i]+1; }

// Second loop to update y
for (i=0; i<n-1; i++)
    { y[i+1]=x[i]+w[i]; }
```

- Outcome: This approach failed because it fundamentally altered the order of calculations. Each y[i+1] needs the corresponding w[i+1] from the same iteration of the original loop, so separating them caused the results to be incorrect.
- Performance: The results showed a drop in GFLOPS and incorrect output due to the disruption of the dependency chain.

GFLOPS Comparison Table

Method	Correctness	GFLOPS (Min)	GFLOPS (Max)
Baseline (Original Loop)	Yes	2.22	2.30
Loop Unrolling	No	1.81	1.83
Loop Splitting	No	N/A	N/A

Conclusion

After testing several optimization methods, we found that vectorization was not feasible for this loop structure due to data dependencies between w[i + 1] and y[i + 1]. Each calculation depends on values from the previous iteration, creating a sequential dependency that prevents safe parallelization.

The most effective solution was to retain the original loop structure and rely on the compiler's general optimizations with the -O3 flag, which provided a slight performance increase while maintaining correctness. This analysis demonstrates that certain dependency-bound loops may not be optimized via typical vectorization or unrolling techniques, highlighting the importance of evaluating dependencies before applying aggressive optimizations.

5. (25 points, Extra credit) The following code in vec5ref.c is not vectorized by clang. Can it be transformed into functionally equivalent code (i.e., satisfies all data dependences) that can be vectorized by clang? If so, create equivalent code in vec5opt.c to achieve higher (single thread) performance. You may use additional data declarations if desired; the only requirement is that sequential performance increases significantly (at least 2x), and the correctness check passes]

```
for (i=0; i<N; i++)
    {sum = 0.0;
    for(j=0; j<N; j++)
        // sum += A[j][i]*A[j][i];
        sum += A[j*N+i]*A[j*N+i];
    x[i] = sum;}
```

Solution.

Background.

The original code in `vec5ref.c` was not vectorized by `clang` due to the presence of nested loops and accumulation operations that created challenges for both vectorization and parallelization. The goal was to transform the code into an optimized version that:

(a) Achieves significant performance improvement (at least 2x).

(b) Maintains correctness as verified by the comparison with the reference implementation.

Approach and Attempts.

We explored several optimization techniques to address the challenges in `vec5ref.c`. Each method focused on improving either correctness or vectorization, but not all attempts succeeded in balancing the two goals. Below, we summarize the attempts and their outcomes:

- Inner-Loop Vectorization: Applied `#pragma omp simd` directives to vectorize the inner loop. While this improved performance, the results were incorrect due to unsafe accumulation across threads.
- Outer-Loop Parallelization: Parallelized the outer loop using `#pragma omp parallel for`, improving correctness but achieving limited vectorization.
- Loop Unrolling: In one attempt, we applied loop unrolling to reduce the overhead of loop control and allow the compiler to identify and optimize multiple iterations at once. Specifically, we unrolled the inner loop by a factor of 4, performing 4 iterations' worth of computations in a single loop body. While this approach improved the performance slightly, the correctness of the results was compromised due to overlapping updates to shared memory locations. The interdependencies between iterations remained, leading to incorrect accumulation of results.
- Blocking and Hybrid Approaches: Introduced blocking with a block size (`B`) to optimize cache utilization while combining parallelization and vectorization. This approach achieved a balance between correctness and performance.

- Loop Tiling with Loop Unrolling: To further enhance data locality and reduce memory bandwidth issues, we combined loop tiling with loop unrolling. This method processed data in smaller chunks (tiles) to improve cache utilization while still leveraging unrolling to reduce loop overhead. Despite better cache behavior, the inter-iteration dependencies in the accumulation process persisted, and vectorization was still limited. Performance gains were marginal compared to blocking alone, and correctness issues were not fully resolved.
- Reduction and Alignment: Used `#pragma omp simd reduction(+:x[:N])` to safely accumulate results across threads and ensure alignment of memory accesses for efficient vectorization. This method provided the best trade-off between correctness and performance.

Final Solution:

The following code represents the best compromise we achieved. It combines blocking, parallelization, and vectorization, ensuring significant performance improvement while minimizing errors:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <immintrin.h>

void vec5_opt(int N, float *__restrict__ A, float *__restrict__ x) {
    const int B = 128; // Block size for better cache utilization
    int i, j, jb;

    // Initialize the result array with zeros
    #pragma omp parallel for schedule(static)
    for (i = 0; i < N; i++) {
        x[i] = 0.0f;
    }

    // Parallelized and vectorized computation
    #pragma omp parallel for schedule(static) private(j, jb)
reduction(+:x[:N])
    for (jb = 0; jb < N; jb += B) {
        int block_end = (jb + B > N) ? N : jb + B;

        for (j = jb; j < block_end; j++) {
            #pragma omp simd aligned(A : 32)
            for (i = 0; i < N; i++) {
                x[i] += A[j * N + i] * A[j * N + i];
            }
        }
    }
```

}

Explanation of Optimizations:

(a) Blocking: Reduces cache misses by processing data in chunks (`B = 128`), ensuring better cache utilization.

(b) Parallelization: The outer loop is parallelized using `#pragma omp parallel for`, distributing workload across threads.

(c) Vectorization: The inner loop uses `#pragma omp simd` for vectorized operations, performing multiple computations simultaneously.

(d) Reduction: Accumulation of results in `x` is handled using `reduction(+:x[:N])`, ensuring correctness in a parallel environment.

(e) Memory Alignment: The `aligned(A : 32)` clause ensures efficient memory access for vectorized operations.

Performance:

- CADE Results:
  - Reference Code: Min: 1.50 GFLOPS; Max: 1.65 GFLOPS
  - Optimized Code: Min: 6.12 GFLOPS; Max: 8.74 GFLOPS
- Lonepeak CHPC Results:
  - Reference Code: Min: 0.91 GFLOPS; Max: 0.92 GFLOPS
  - Optimized Code: Error: 941 differences found over the threshold of 0.000000

Conclusion:

Despite significant efforts to optimize the code using vectorization, blocking, loop tiling, and other methods, the optimized code still produces errors compared to the reference implementation. While performance increased compared to non-vectorized attempts, the correctness of results remains a limitation due to inter-iteration dependencies.

Despite our best efforts, achieving both full correctness and optimal vectorization was not feasible. The dependency between iterations in the inner loop accumulation posed challenges for correctness in certain configurations. The final solution represents the best trade-off, achieving significant performance improvement and minimizing errors.

Based on the outcomes of various approaches, we conclude that this specific code is unlikely to achieve simultaneous full vectorization and correctness under the current constraints. This highlights the challenges of optimizing such dependency-heavy loops for modern compilers.