Spcc Case Study

- **Analyze the impact of LLVM (loop) optimizations on the execution time of program**
- **Analyze the impact of loop optimizations on matrix multiplication performance**

Mark Lopes (9913)
Jonathan Gomes (9900)
Sanyo Fonseca (9896)
Allan Monis (9920)
Harshada Gawas (10401)

Loop Unrolling

- **Reduces the loop iterations by processing multiple elements per iteration**

<table>
<tr><th>Before Optimization</th><th>After Optimization</th></tr>
<tr><td>

```c
void multiplyByTwo(int* arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] *= 2;
    }
}
```

</td><td>

```c
void multiplyByTwo(int* arr, int n) {
    int i = 0;
    for (; i <= n - 4; i += 4) {
        arr[i] *= 2;
        arr[i + 1] *= 2;
        arr[i + 2] *= 2;
        arr[i + 3] *= 2;
    }
    for (; i < n; i++) {
        arr[i] *= 2;
    }}
```

</td></tr>
</table>

Benefits of Optimization

- ✅ **Fewer loop control operations** (less incrementing, fewer condition checks)
- ✅ **Fewer branch instructions**, reducing CPU pipeline stalls

Trade-offs

- ❌ **Increased code size** (more instructions written explicitly)
- ❌ **Reduced flexibility** (harder to handle dynamic loop bounds efficiently)

Loop Vectorization

- **Optimizes loops by using SIMD instructions, enabling the CPU to process multiple elements in parallel, improving performance on large datasets.**

Before Optimization	After Optimization

<pre>void multiplyByTwo(float* arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] *= 2.0f;
    }
}</pre>
	<pre>#include <immintrin.h> // For AVX
	
	void multiplyByTwo(float* arr, int n) {
	    int i = 0;
	    for (; i <= n - 8; i += 8) {
	        __m256 v = _mm256_loadu_ps(&arr[i]);
	        v = _mm256_mul_ps(v, _mm256_set1_ps(2.0f));
	        _mm256_storeu_ps(&arr[i], v);
	    }
	    for (; i < n; i++) {
	        arr[i] *= 2.0f;
	    }
	}</pre>

Optimized Performance ✅ Processes 8 elements at a time (AVX-256) instead of just 1 ✅ Reduces loop iterations and overhead	**Performance Issues** ❌ Only one element is processed per iteration ❌ More loop overhead (loop condition checks, increments, memory accesses)

Loop Fusion

- **Merges two loops iterating over the same range to reduce memory accesses and improve data locality.**

Before Optimization O(2n)	After Optimization O(n)

`for (int i = 0; i < N; i++) {  // Loop 1`
`    A[i] = B[i] + 2;`
`}`
`for (int i = 0; i < N; i++) {  // Loop 2`
`    C[i] = A[i] * 3;`
`}`
	`for (int i = 0; i < N; i++) {  // Single Loop`
	`    A[i] = B[i] + 2;`
	`    C[i] = A[i] * 3;`
	`}`

🎯 *Automata Concept Used:*

- *State Minimization in FSMs – Just like merging equivalent states in FSMs to simplify computation, we merge loops to reduce execution overhead.*

Loop Invariant Code Motion (LICM)

- **Moves invariant computations (independent of the loop variable) outside the loop to avoid redundant calculations.**

Before Optimization O(n)	After Optimization O(n)

Before Optimization O(n)

```
for (int i = 0; i < N; i++) {
    int x = 10 * 5;  // Computed in every iteration
    A[i] = x + i;
}
```

After Optimization O(n)

```
int x = 10 * 5;  // Computed only once (
for (int i = 0; i < N; i++) {
    A[i] = x + i;
}
```

🎯 *Automata Concept Used:*

- *Reduction of Stack Operations in PDAs – LICM reduces redundant calculations just like optimizing push/pop operations in PDAs.*

Loop Unswitching

- **An optimization that moves loop-invariant conditional statements outside the loop to reduce checks and improve performance.**

Before Optimization

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i][j] = 0;
        for (int k = 0; k < n; k++) {
            if (flag) C[i][j] += A[i][k] * B[k][j];
            else C[i][j] -= A[i][k] * B[k][j];
        }
    }
}
```

After Optimization

```
if (flag) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
} else {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                C[i][j] -= A[i][k] * B[k][j];
}
```

Loop Interchange

- **Swaps nested loops to improve memory locality and cache performance.**

Before Optimization	After Optimization

`void matrix_multiply(int n) {`
`    int A[n][n], B[n][n], C[n][n];`
`    for (int i = 0; i < n; i++)`
`        for (int j = 0; j < n; j++)`
`            for (int k = 0; k < n; k++)`
`                C[i][j] += A[i][k] * B[k][j];`
`}`
	`void matrix_multiply(int n) {`
	`    int A[n][n], B[n][n], C[n][n];`
	`    for (int i = 0; i < n; i++)`
	`        for (int k = 0; k < n; k++)`
	`            for (int j = 0; j < n; j++)`
	`                C[i][j] += A[i][k] * B[k][j];`
	`}`

Loop Peeling

Loop peeling extracts the first few iterations of a loop outsidefrom the main loop body.

Before Loop Peeling O(N)

```
for (int i = 0; i < N; i++) {
    A[i] = B[i] * 2;
}
```

Optimization

After Loop Peeling O(N-1) ≈ O(N)

```
A[0] = B[0] * 2;
for (int i = 1; i < N; i++) {
    A[i] = B[i] * 2;
}
```

🚀 Summary

Factor	Before Peeling	After Peeling
Time Complexity	O(N)	O(N)
Extra Condition Checks?	✅ Yes (slower)	❌ No (faster)
Cache & Memory Alignment	⚠️ May be unoptimized	✅ More efficient
CPU Performance	🔴 Some branch misprediction	🟢 More predictable execution
Overall Speedup	⌛ Normal	🚀 ~5-10% faster

Loop Distribution

- Loop Splitting breaks a single loop into multiple smaller loops.
- This improves parallelism, cache efficiency, and CPU register usage.

Before Loop Splitting

```
for (int i = 0; i < N; i++) {
    A[i] = B[i] + C[i];
    D[i] = E[i] * F[i];
}
```

After Loop Splitting

```
for (int i = 0; i < N; i++) {
    A[i] = B[i] + C[i];
}

for (int i = 0; i < N; i++) {
    D[i] = E[i] * F[i];
}
```

Impact of loop optimizations on matrix multiplication performance

- **Loop Unrolling**-Reduces iterations, increases instruction parallelism.
- **Loop Interchange**-Swaps loops, enhances memory locality.

Loop Unrolling

```c
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < N; k++) {  // Each
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

→

```c
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < N; k += 2) {  // Unrolling factor of 2
            C[i][j] += A[i][k] * B[k][j] + A[i][k+1] * B[k+1][j];
        }
    }
}
```

Optimization	Loop Iterations	Operations per Iteration	Total Multiplications	Execution Speed
Without Unrolling	1000 × 1000 × 1000	1	1 Billion	⏳ Slow
Unrolling (Factor of 2)	1000 × 1000 × 500	2	1 Billion	🚀 Faster

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

Loop Interchange

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

→

```
for (int i = 0; i < N; i++) {
    for (int k = 0; k < N; k++) {
        for (int j = 0; j < N; j++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
Matrix C:
  1  2  3
  4  5  6
  7  8  9
```

Memory layout in row-major order:
[1, 2, 3, 4, 5, 6, 7, 8, 9] (Stored row-wise)

Row-wise Access

1️⃣ CPU requests 1 → Cache is empty → Fetch from RAM (RAM loads 1,2,3 into the cache).
2️⃣ CPU requests 2 → Cache Hit! (Data is already there).
3️⃣ CPU requests 3 → Cache Hit! (Data is still there).

Column-wise Access

1️⃣ CPU requests 1 → Cache is empty → Fetch from RAM (RAM loads 1,2,3 into the cache).
2️⃣ CPU requests 4 → Cache Miss! (Loads 4,5,6, but we only need 4 now).
3️⃣ CPU requests 7 → Cache Miss! (Loads 7,8,9, but we only need 7).