Friedrich-Alexander-Universität
Technische Fakultät

Computer Architecture Practical Exercise

3 Loop Unrolling

Kenan Gündogan[1] Philipp Gündisch[1]

[1]Friedrich-Alexander Universität Erlangen-Nürnberg, Chair of Computer Science 3 (Computer Architecture)

November 18, 2024

Motivation

Compiler Optimization

- Compilers implicitly perform code optimization techniques
- With `icx -O3 -xHost` aggressive optimization for the specific host machine is requested
- Two very effective optimization techniques are called:
  - **Loop Unrolling** (this week)
  - **Vectorization** (next week)
- We will implement these techniques manually to understand their implications on the performance

Disable Compiler Optimizations

Before implementing these optimization techniques we need to disable the implicit compiler optimizations. This can be done with compiler directives (`#pragma`) and compiler flags.

- Disable **Loop Unrolling** with `#pragma nounroll`
- Disable **Vectorization** by
  - the use of `#pragma novector`,
  - replacing `-O3` with `-O1` in compiler flags
  - and removing `-xHost` from compiler flags

Example applied to `vec_sum()`:

```c
#include "vec_sum.h"

float vec_sum(const float * restrict array, int32_t length) {
        float sum = 0.0f;
        #pragma nounroll
        #pragma novector
        for(int32_t i = 0 ; i < length ; i++) {
                sum += array[i];
        }
        return sum;
}
```

Loop Unrolling

Introduction

Loops can be manually unrolled as displayed in the following example.

```
#include "vec_sum.h"

float vec_sum(const float * restrict array, int32_t length) {
        float sum[2] = { 0.0f, 0.0f };
        int32_t remainder = length % 2;

        #pragma nounroll
        #pragma novector
        for(int32_t i = 0 ; i < length-remainder ; i+=2) {
                sum[0] += array[i];
                sum[1] += array[i+1];
        }

        sum[0] += sum[1];
        if(remainder)
                sum[0] += array[length-1];
        return sum[0];
}
```

Loop Unrolling

Assembler Code

Loop Unrolling reduces the control flow overhead of loops.
Assembler code resembling the for loop `vec_sum` without loop unrolling.

```
.LBB0_4:
        addss   (%rdi,%rcx,4), %xmm0
        incq    %rcx
        cmpq    %rcx, %rax
        jne     .LBB0_4
```

Same code with twofold loop unrolling.

```
.LBB0_7:
        addss   (%rdi,%rdx,4), %xmm0
        addss   4(%rdi,%rdx,4), %xmm1
        addq    $2, %rdx
        cmpq    %rcx, %rdx
        jl      .LBB0_7
```

Note: the assembler codes were produced with:

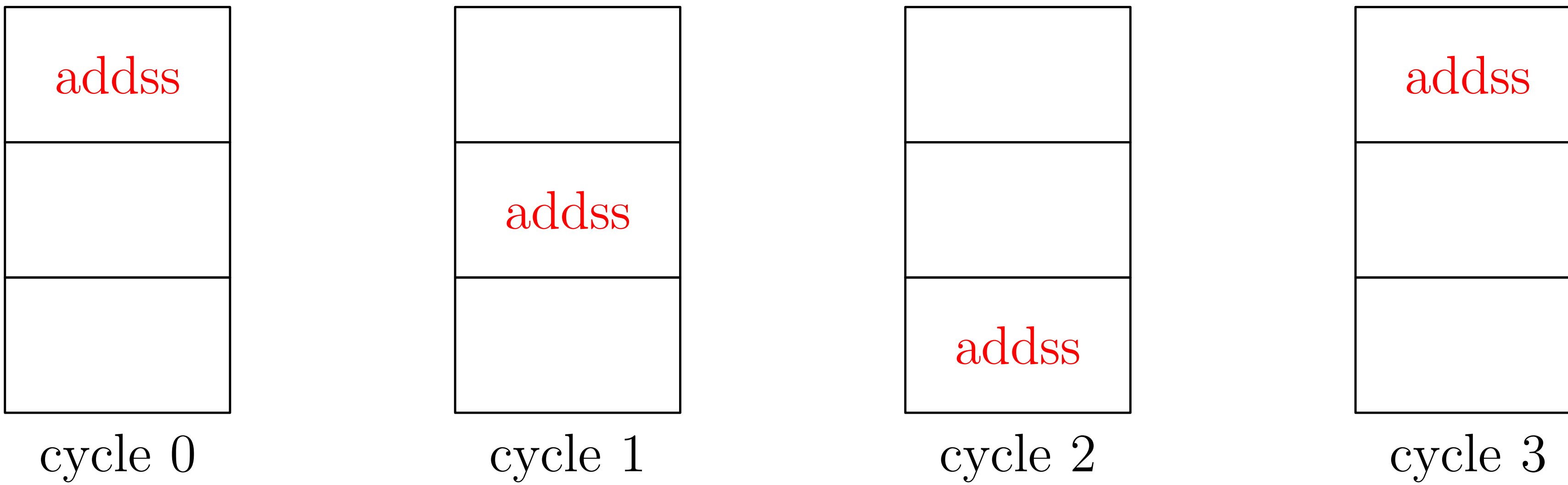
`icx -I ./include/ -S -O1 vec_sum.c`

Loop Unrolling

No Unrolling - Effects on Pipelined Units

```c
#pragma novector
#pragma nounroll
for(i = 0; i < n; i += 1) {
        sum += A[i];
}
```

```asm
.LBB0_4:
        addss   (%rdi,%rcx,4), %xmm0
        incq    %rcx
        cmpq    %rcx, %rax
        jne     .LBB0_4
```



Loop Unrolling

2-fold Unrolling - Effects on Pipelined Units

```c
#pragma novector
#pragma nounroll
for(i = 0; i < n; i += 2) {
        sum0 += A[i];
        sum1 += A[i+1];
}
```

```asm
.LBB0_7:
        addss   (%rdi,%rdx,4), %xmm0
        addss   4(%rdi,%rdx,4), %xmm1
        addq    $2, %rdx
        cmpq    %rcx, %rdx
        jl      .LBB0_7
```

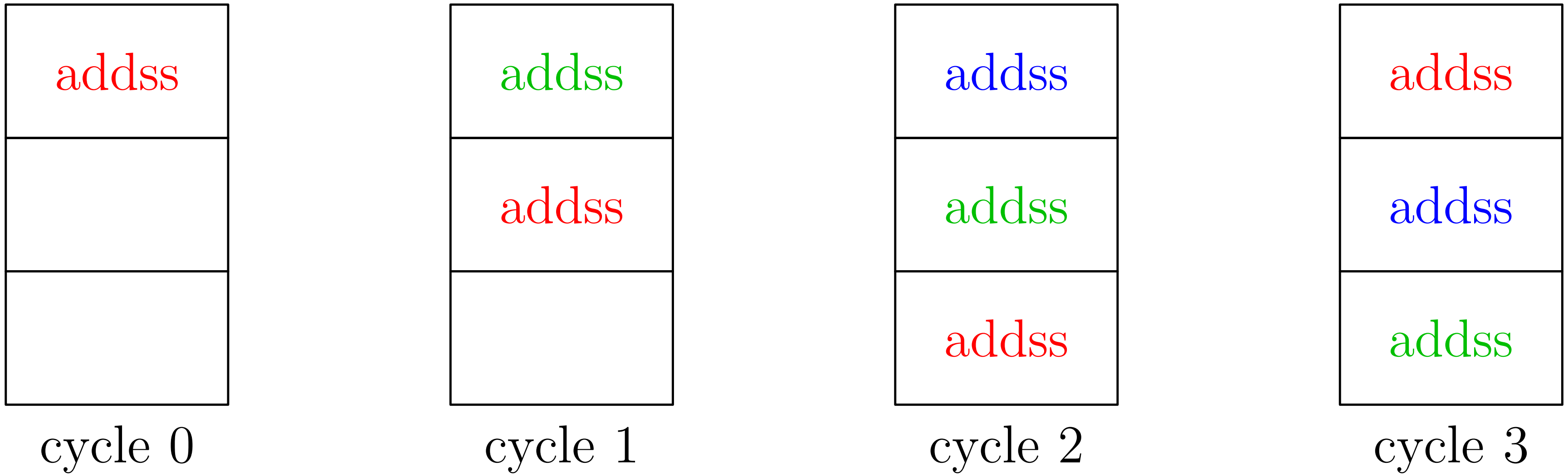
cycle 0	cycle 1	cycle 2	cycle 3
addss	addss		addss
	addss	addss	
		addss	addss

Loop Unrolling

3-fold Unrolling - Effects on Pipelined Units

```c
#pragma novector
#pragma nounroll
for(i = 0; i < n; i += 3) {
        sum0 += A[i];
        sum1 += A[i+1];
        sum2 += A[i+2];
}
```

```asm
.LBB0_8:
        addss   (%rdi,%rdx,4), %xmm1
        addss   4(%rdi,%rdx,4), %xmm2
        addss   8(%rdi,%rdx,4), %xmm0
        addq    $3, %rdx
        cmpq    %rcx, %rdx
        jl      .LBB0_8
```



Task 3.1: VecSum Loop Unrolling

Implementation

- Adapt `vec_sum.c` to suppress unrolling and vectorization.
- Adapt the loop of `vec_sum()` to manually unroll till saturation.
- Benchmark each implementation for 1KiB - 32MiB (1 second runtime)
- Benchmark time efficiently (10m max.) until the saturation is found.
- Once unrolling saturation is identified, design a benchmark to proof the results (1h max.) Hint: You do not need to implement all consecutive unrolling variations as long as you can proof the correct saturation. Hint: Implement solutions in separate files or use `#ifdef`.

Task 3.2: Jacobi Loop Unrolling

Implementation

- Adapt `jacobi.c` to suppress unrolling and vectorization.
- Adapt the innermost loop of `jacobi()` to support benchmarking of:
  - No unrolling
  - (OPTIONAL) Twofold unrolling
  - Fourfold unrolling
  - Eightfold unrolling

  Hint: Implement solutions in separate files or use `#ifdef`.
- Benchmark each implementation for 1KiB - 128MiB (1 second runtime)
- Why does jacobi unrolling not show the same performance gains as vec_sum?

Task 3.3: Plot Benchmark Results

Visualization

Visualize the benchmark results with the tool of your choice:

- Plot Jacobi and VecSum into different plots
- Draw a line chart with each line resembling an unrolling variant
- Choose the memory consumption as the X-axis
- Choose the performance metrics for the Y-axis (AdditionsPerSecond, MUp/s)

Task Overview

- E 3.1: VecSum Loop Unrolling
  - No unrolling
  - Unrolling till saturation
  - Hint: No unroll 16 kiB $\rightarrow 600 \cdot 10^6$ Adds/s
  - Hint: Fourfold unroll 16 kiB $\rightarrow 2500 \cdot 10^6$ Adds/s
- E 3.2: Jacobi Inner Loop Unrolling
  - No unrolling
  - (OPTIONAL) Twofold unrolling
  - Fourfold unrolling
  - Eightfold unrolling
  - Hint: no unroll 50x50 grid $\rightarrow 550 \cdot 10^6$ Updates/s
  - Hint: fourfold unroll 50x50 grid $\rightarrow 800 \cdot 10^6$ Updates/s
- E 3.3: Benchmark Result Plotting
  - One VecSum plot comparing the unrolling variants
  - One Jacobi plot comparing the unrolling variants

Appendix: `#define`

Best Practice

Defines can be seen as compile time variables. They are evaluated by the preprocessor of the compiler. Since the compiler knows their values, optimizations can be applied during the compilation process. This is a huge advantage over standard C variables.

```
#ifndef NUMBER                         // Check if it was previously defined
        #define NUMBER (3+4)           // Hint: Always use braces
#endif

// Use of conditionals
#if NUMBER == 7
        printf("NUMBER IS 7\n");
#elif NUMBER > 7
        printf("NUMBER IS GREATER THAN 7\n");
#else
        printf("NUMBER IS LESS THAN 7\n");
#endif
```

Defines can be passed to the compiler with `-D`:

`icx -DNUMBER=7 ...`

Appendix: `Sum Formula`

Result Validation

It is important to validate the benchmark results to ensure no work is skipped and the benchmark is still valid after optimization.

- During optimization, code complexity increases and errors can be introduced.
- Compiler optimizations `-O3` may change code behavior in an unexpected way.

For the `vec_sum()` validation the following sum formula can be used. Note that the `vec_sum()` function works with 32 bit floating point precision and is not numerical accurate for very large sums!

$$\sum_{k=1}^{n} = \frac{n \cdot (n+1)}{2}$$

For the `jacobi()` validation the `draw_grid()` function should be used and visually checked. Alternatively, you can also compare the results (same grid size, same number of full grid updates) of two different implementations. Ensure to consider floating point accuracy errors when comparing the results.

Appendix: Checklist

Performance Optimization

During the timeline of this class new bullet points will be added.

- Compiling
  - Choice of the compiler (`icx`)
  - Compiler flag to optimize aggressively (e.g. `-O3`)
  - Compiler flag to adapt for specific hardware (e.g. `-xHost`)
- Programming Techniques (if applicable)
  - Use `#define` and `const` instead of variables
  - Data type aware programming
  - Use aligned memory (e.g. with `_mm_malloc()` or `posix_memalign()`)
  - Consecutive address iteration
- Measurement
  - Reasonable benchmark time
  - Reasonable benchmark workload
  - Reduce interference factors to a minimum