Computer Architecture Practical Exercise

4 Vectorization

Kenan Gündogan[1] **Philipp Gündisch**[1]

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

November 11, 2024

Motivation

SIMD

$$G_{x,y}^{i+1} = \frac{G_{x,y+1}^{i} + G_{x,y-1}^{i} + G_{x+1,y}^{i} + G_{x-1,y}^{i}}{4}$$

- Jacobi performs the same operation over a huge number of grid cells
- There are no data dependencies between the operations (within a time step simulation)

$\rightarrow$ SIMD (Single Instruction Multiple Data) techniques can be applied.

Larger registers paired with specialized instructions (*intrinsics*) can be used to batch process multiple numbers at once.

SIMD

Vectorization

In contrast to scalar instructions, SIMD instructions execute the same instruction on vectors. The Intel x86 architecture (as a CISC) offers specific instruction set extensions:

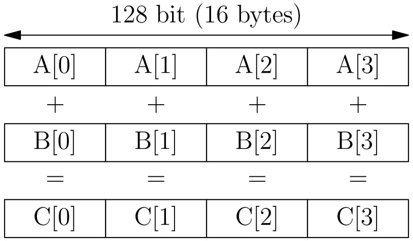
- **SSE** (Streaming SIMD Extension)
  - **128 bit** instructions
  - 4 `float` values per instruction
  - 2 `double` values per instruction
- **AVX** (Advanced Vector Extension)
  - **256 bit** instructions
  - 8 `float` values per instruction
  - 4 `double` values per instruction
- **AVX-512** (Advanced Vector Extension 512 Bit)
  - **512 bit** instructions
  - 16 `float` values per instruction
  - 8 `double` values per instruction
- many more ...

Note: these extensions offer further operations for other data types as well.

SIMD

Vectorization

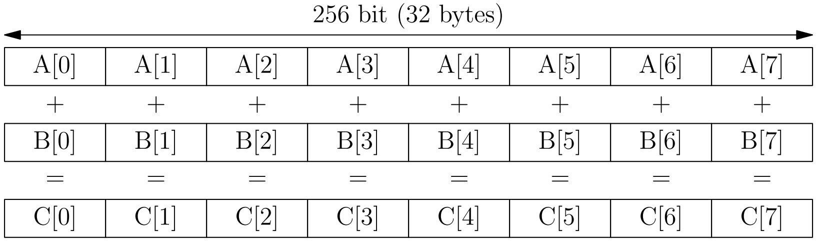
- **SSE** (Streaming SIMD Extension)



128 bit (16 bytes)

A[0]	A[1]	A[2]	A[3]
+	+	+	+
B[0]	B[1]	B[2]	B[3]
=	=	=	=
C[0]	C[1]	C[2]	C[3]

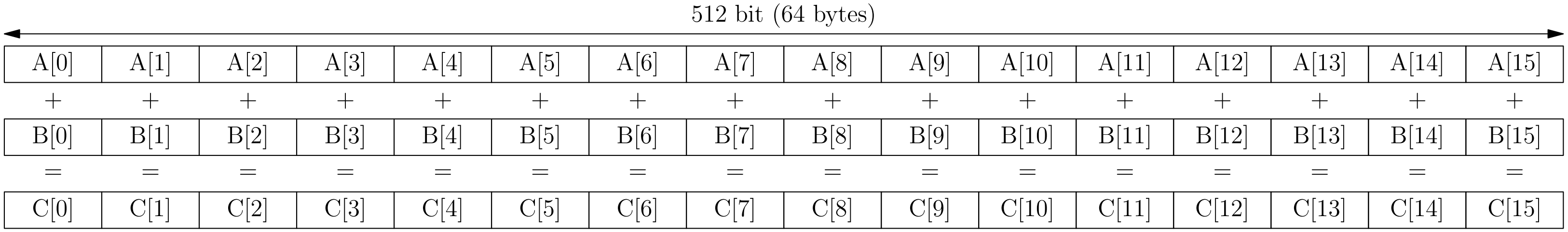
- **AVX** (Advanced Vector Extension)



256 bit (32 bytes)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
+	+	+	+	+	+	+	+
B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]
=	=	=	=	=	=	=	=
C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]

- **AVX-512** (Advanced Vector Extension)



512 bit (64 bytes)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]	B[10]	B[11]	B[12]	B[13]	B[14]	B[15]
=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]	C[10]	C[11]	C[12]	C[13]	C[14]	C[15]

Intel Intrinsics

API

Use Intel's official intrinsics documentation to identify relevant functions.
To make the intrinsic functions available in C you need to include the header:

```
#include <immintrin.h>
```

Example for the naming convention of intrinsic functions: `_mm256_add_pd`

- Identification of the instruction set
  - SSE with `mm`
  - AVX with `mm256`
  - AVX-512 with `mm512`
- Identification of the operation type with
  - `load`, `store`, `add`, `mul`, `div`, ...
- Identification of scalar vs SIMD with
  - scalar with `s`
  - packed (SIMD) with `p`
- Identification of the data type with
  - `float` (4 bytes) with `s`
  - `double` (8 bytes) with `d`

Note: You need to ensure the processor supports the dedicated instruction set extension.

Vectorization

Introduction

Unrolled loops can be manually vectorized as displayed in the following example.

```
#include <immintrin.h>
#include "vec_sum.h"

float vec_sum(const float * restrict array, int32_t length) {
        float    sum       = 0.0f;
        int32_t  remainder = length % 8;
        __m256   result    = _mm256_setzero_ps();
        __m256   accu;

        #pragma nounroll
        #pragma novector
        for(int32_t i = 0 ; i < length-remainder ; i+=8) {
                accu   = _mm256_loadu_ps(&array[i]);
                result = _mm256_add_ps(result, accu);
        }

        // Add result and remainder elements into ’sum’
        ...
        return sum;
}
```

Vectorization

Assembler Code

Assembler code resembling the for loop of `vec_sum()` without optimization.

```
..B1.5:
        addss    (%rdi,%rax,4), %xmm0
        incl     %eax
        cmpl     %esi, %eax
        jb       ..B1.5
```

Assembler code resembling the eightfold unrolled loop with SIMD instructions.

```
..B1.5:
        vaddps   (%r8,%rax,4), %ymm0, %ymm0
        addl     $8, %eax
        cmpl     %edi, %eax
        jb       ..B1.5
```

Note: the assembler codes were produced with:

```
icx -I ./include/ -S -O3 vec_sum.c
```

Task 4.1: Jacobi SSE & AVX

Implementation & Visualization

- Keep suppressing implicit unrolling and vectorization
- Keep the naive implementation without unrolling
- Adapt the twofold unrolled loop to use SSE instructions
- Adapt the fourfold unrolled loop to use AVX instructions
- Adapt the eightfold unrolled loop to use AVX512 instructions
- Benchmark all three implementations for 1KiB - 128MiB (1 second runtime)
- Draw a line chart with the tool of your choice
- Choose the memory consumption as the X-axis
- Choose the performance metrics for the Y-axis (MUp/s)
- **NOTE: USE UNALIGNED LOADS**

Optional Task 4.2: VecSum

NOT MANDATORY

Beat the compiler by exceeding the performance of e00! (for at last one measurement)

- Keep suppressing implicit unrolling and vectorization
- Adapt the eightfold unrolled loop to use AVX instructions for summation
- Keep the other implementations from previous exercise
- Implement a twofold, threefold and fourfold unrolling for the AVX instructions by replacing `#pragma nounroll` with `#pragma unroll (N)`
- Benchmark all three implementations for 1KiB - 32MiB (1 second runtime)
- Draw a line chart with the tool of your choice
- Choose the memory consumption as the X-axis
- Choose the performance metrics for the Y-axis (AdditionsPerSecond)
- **NOTE: USE UNALIGNED LOADS**

Task Overview

- E 4.1: Jacobi SSE & AVX
  - Update twofold unrolling with SSE instructions
  - Update fourfold unrolling with AVX instructions
  - Compare results in a linechart
- Optional 4.2: VecSum AVX
  - Update eightfold unrolling with AVX instructions
  - Unroll the eightfold unrolled with AVX instructions further
  - Compare results in a linechart

Appendix: CPU Flags

Identify Supported Extensions

There are multiple ways to identify the instruction set extensions of your processor.

- `cat /proc/cpuinfo`
- `lscpu`
- Internet search for the processor model

Appendix: Checklist

Performance Optimization

During the timeline of this class new bullet points will be added. Recently added entries are bold.

- 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
  - **Variable declarations outside of loops**
  - **Reduce function calls**
  - **Use intrinsics (to utilize SIMD)**
- Measurement
  - Reasonable benchmark time
  - Reasonable benchmark workload
  - Reduce interference factors to a minimum