Tuning CPU Performance:

Introduction to SIMD Optimization

Mainack Mondal
Sandip Chakraborty

CS 60203
Autumn 2024

Outline

- CPU Performance Optimization
- Motivation
- SIMD Overview
    - What is SIMD?
    - Data Types in SIMD Programming
    - Instructions
    - Example

CPU Performance Optimization

(Slides partially taken from Marat Dukhan)

Components of CPU Performance

Peak FLOPs =

Number of Cores × ← Task Level Parallelism

FLOPs per Instruction × ← SIMD and FMA

Instructions per cycle × ← Instruction Level Parallelism

Cycles per second ← Frequency

CPU Optimization 101

- Task-Level Parallelism (across cores)
  - Cilk, Cilk++
  - OpenMP
- Instruction-Level Parallelism
  - Reordering
  - Out-of-Order Execution
  - Speculative Execution
  - Branch Prediction
- SIMD

Motivation

(Slides partially taken from Lukas Pietzschmann)

Motivation

Lets see an example code:

```
void mul4(float* arr) {
   for(int i=0; i < 4; ++i) {
      const float f = arr[i];
      arr[i] = f * f;
   }
}
```

Problems ?

Why is it bad?

- Short Loops are bad. Why?
  - Branch Prediction fail often
- Unnecessary extra instructions
  - Many load/store Instructions
  - Unnecessary add instructions

Lets Make It Better

But how? **`Unroll Loops`**

```
void mul4(float* arr) {
   arr[0] = arr[0] * arr[0];
   arr[1] = arr[1] * arr[1];
   arr[2] = arr[2] * arr[2];
   arr[3] = arr[3] * arr[3];
}
```

Problems ?

Why is it good ?

- No branches to predict
- No loops

Why is it bad ?

- Bad Machine Code
- Too many load/store instructions

Can we do even better ?

Making It Even Better

🛸 Enters SIMD 🛸

```
void mul4(float* vec) {
   __m128 f = _mm_loadu_ps(vec);
   f = _mm_mul_ps(f, f);
   _mm_storeu_ps(vec, f);
}
```

Assembly
(approximate)

```
mul4:
    movups  xmm0, XMMWORD PTR [rdi]
    mulps   xmm0, xmm0
    movups  XMMWORD PTR [rdi], xmm0
    ret
```

Why is it even better ?

- No loops
- No branches to predict
- Nice machine code
- We square all floats at once

Performance

TestBench: Dot Product of two vectors, each of size 256,000

Description	Time (in µs)
Regular floating point math	439
SSE dpps instruction	181
AVX vdpps instruction	103

On an average:

- SSE: 2.5x speed increase
- AVX: 4x speed increase

Credits: Improving performance with SIMD intrinsics in three use cases

Outline

- CPU Performance Optimization

- Motivation

- **SIMD Overview**
    - What is SIMD?
    - Data Types in SIMD Programming
    - Instructions
    - Example

SIMD Overview

What is SIMD?

SIMD : **S**ingle **I**nstruction **M**ultiple **D**ata

Comes from *Flynn's Taxonomy* of types of Computing Systems:

	Single Data Stream	**Multiple Data Stream**
Single Instruction	SISD: Intel Pentium 4	SIMD: SSE/AVX in x86
Multiple Instruction	MISD: No examples	MIMD: Intel Xeon Phi

SIMD in C/C++

Intrinsics:

- Usually implemented “inside” the computer.
- Allow for better optimisations than raw inline assembly
- Provide access to instructions that cannot be generated using the standard constructs

Compiler Support for SIMD in C/C++

- Compiler provides options like **`-march=corei7`** (gcc/clang)
- Provides two main functions:
  - maps directly to extended assembly instructions upto SSE4.2
  - allows the compiler to optimize programs using these instructions

Auto-Vectorization:

- Compiler automatically uses these instructions for optimization
- Ever wondered what happens when you use the “**`-O3`**” flag
  - Compiler tries for auto-vectorization (there is a catch)

SIMD Data Types

SSE2 →

	16 bytes	32 bytes
32 bit float	`__m128`	`__m256`
64 bit double	`__m128d`	`__m256d`
32/64 bit integer	`__m128i`	`__m256i`

- CPU doesn’t distinguish between `__m128, __m128d` and `__m128i`
  - This information is only used for type checking

- Compiler automatically assigns the values to registers
  - [Caution] Only 16 (8+8) registers underneath the compiler (Why caution?)

SIMD Instructions: Loading From Memory

```
void mul4(float* vec) {

   __m128 f = _mm_loadu_ps(vec);

   f = _mm_mul_ps(f, f);
   _mm_storeu_ps(vec, f);
}
```

We can load:

- four values aligned
- four values unaligned
- four values in reverse

…

Arithmetic Operations

Examples of arithmetic operations:

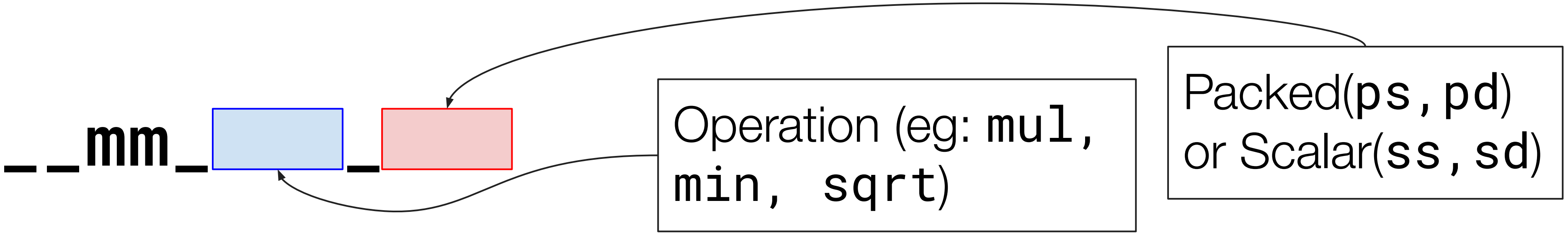
- `__mm_mul_ps`
- `__mm_add_ps`
- `__mm_min_ps`

```
void mul4(float* vec) {
    __m128 f = _mm_loadu_ps(vec);

    f = _mm_mul_ps(f, f);

    _mm_storeu_ps(vec, f);
}
```

In general, such instructions have the following structure:



`__mm_` ▭ **`_`** ▭

Operation (eg: `mul`, `min`, `sqrt`)

Packed(`ps`, `pd`) or Scalar(`ss`, `sd`)

Storing To Memory

```
void mul4(float* vec) {
   __m128 f = _mm_loadu_ps(vec);
   f = _mm_mul_ps(f, f);

   _mm_storeu_ps(vec, f);
}
```

We can store:

- four values aligned
- four values unaligned
- four values in reverse

…

An Example

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

Header to be included

```
float* add(const float* a, const float* b, size_t size) {
   float* result = new float[size];
   const auto numof_vectorizable_elements = size - (size % 4);
   unsigned i = 0;
   for (; i < numof_vectorizable_elements; i += 4) {
      __m128 a_reg = _mm_loadu_ps(a + i);
      __m128 b_reg = _mm_loadu_ps(b + i);
      __m128 sum = _mm_add_ps(a_reg, b_reg);
      _mm_storeu_ps(result + i, sum);
   }
   for (; i < size; ++i)
      result[i] = a[i] + b[i];
   return result;
}
```

Compile with flags:
`-mavx` or `-mavx2`

But… Where do we use SIMD?

The simple answer is : Where the performance of your program is dependent on CPU

Example:

- Cryptographic Computations
  - SHA Computations, Elliptic Curve Operations
- Graphics
  - Processing 3D graphics, audio/video etc.
- Machine Learning
  - Neural Networks
  - Image Processing
  - …

and many more …