

Introduction to HPC

Lecture 3

Jakub Gałecki



Front end – recap

Back end overview, ILP

SIMD

Vector registers and instructions

Compiler intrinsics

Leveraging SIMD-aware libraries: Eigen



The front end is the part of the CPU which fetches and decodes instructions and submits them to be executed by the back end

To use an (imperfect) analogy: the front end is the waiter, the back end is the cook

Consists of:

- L1I cache
- Fetch unit
- Decode unit
- Branch predictor

• ...



Explain how a CPU can execute 16 DPFlops *per cycle* (newer ones can do 32)

Complete our mental model of the CPU

Show that a single core has significant parallelism

See how to leverage this in practice

CPU back end - an overview



The back end executes the instructions (μ ops) emitted by the front end

Modern CPUs are **superscalar**: they can emit and execute more than 1 instruction per cycle

We refer to the circuits responsible for actually executing the instructions as **execution units**

Execution units:

- independent Instruction Level Parallelism
- each one can do a subset of operations
- \cdot may be pipelined (depending on the operation)



Out-of-order execution:

- execution order is dynamically determined by the scheduler
- \cdot goal: saturate the execution units
- has to maintain correctness

Register renaming:

reduces data hazards

Store buffer:

 \cdot helps avoid stalls when waiting for writes to complete

CPU block diagram example





Source: https://en.wikichip.org/w/images/7/7e/skylake_block_diagram.svg

SIMD



SIMD == Single Instruction Multiple Data

It is very often the case that we want to perform the same operation on many pieces of data (e.g. process an array). It would greatly speed up our program if we could process it in chunks, not element by element.





Consider the following code:

int a = 0b0101, b = 0b0001; int c = a ^ b; // 0b0100

Bitwise **xor** on **a** and **b** does 32 **xor**s on the bits of these integers. How many instructions does it take the CPU?

xor eax, esi

SIMD generalizes of this concept to more complex types It is actually a very old idea (Cray-1) CPU + SIMD == vectorization



What do we need to make this happen?

- Vector registers (e.g. ymm0 ymm15)
- Vector execution units (see block diagram)
- Vector instructions (see next slide)

It would also be great to have compiler support and some way to access vector operations without explicitly writing assembly code

SSE (Streaming SIMD Extension) have been around for over 20 years, with further extensions (SSE[2, 3, 4], AVX, AVX2, AVX-512, others...) being added.



Name	Explanation
vmovapd	move aligned packed double
vmovupd	move unaligned packed double
vaddpd	vectorized 256b add
vmulpd	vectorized 256b multiply
vfmadd132pd	vectorized fused multiply-add
vpermpd	permute (shuffle) values in 256b register
vmovddup	duplicate even-indexed elements

Live demo





https://godbolt.org/z/5bxbqe85M



Data layout matters

SoA: using Points = vector< array< double, 3 > > SoA: using Points = array< vector< double >, 3 > To leverage SIMD, we need to load in contiguous chunks of data For SIMD, SoA > AoS

The eternal struggle of performance vs. maintainability



Getting the compiler to generate the optimal vectorized assembly can be tricky

- \cdot we may know more than the compiler
- compilers aren't perfect (and that's ok)

Compiler intrinsics bridge the gap between C(++) and assembly

- C types representing vector registers
- \cdot C functions which directly map to assembly instructions
- Transparent to the compiler optimization is possible

Live demo







- What we did up until now was fairly platform-specific
- Different CPU architectures have different vector extensions
- Automatic vectorization can work better or worse, depending on the compiler
- Now that we understand what's happening under the hood, let's take a step towards more easy-to-use, cross-platform solutions...

Eigen



Eigen is a C++ linear algebra library

- Combines a friendly, Matlab-like syntax with the performance of C++
- Vectorized and optimized for most platforms under the hood
- Thread-parallelized (though we will not use this capability just yet)
- Header only (no installation required)
- Code + docs: https://eigen.tuxfamily.org/dox/
- Side note: some GPU support



```
#include <iostream>
#include <Eigen/Dense>
```

```
int main()
{
  Eigen::MatrixXd m(2,2);
  m(0,0) = 3;
  m(1,0) = 2.5;
  m(0,1) = -1;
  m(1,1) = m(1,0) + m(0,1);
  std::cout << m << '\n';</pre>
}
```



Modern CPUs can execute more than 1 instruction per cycle

- We can leverage vectorization to significantly speed up the processing of arrays of data
- SoA > AoS (for SIMD)
- There are different ways of accessing SIMD capabilities
- We can leverage SIMD-aware libraries (e.g. Eigen) to get the performance without the messy code



$\rightarrow\,$ CPUs are highly parallel within a single core

- → High-performance software must be tailored to the hardware
- → Leverage libraries, don't hand-code



