NTNU – Trondheim
Norwegian University of
Science and Technology

Roofline analysis

Jan.Christian.Meyer@ntnu.no

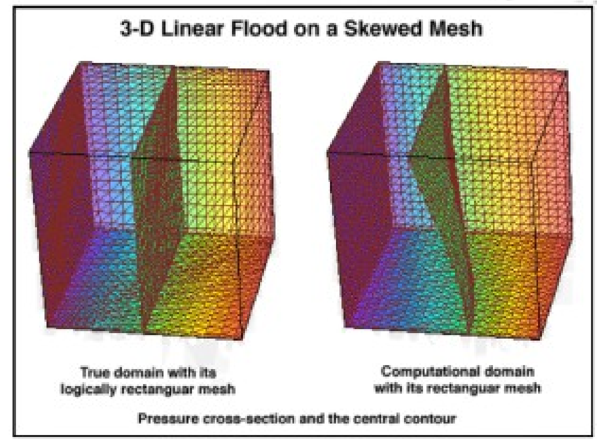
Way back in the beginning...

- We started the semester with talking about the von Neumann computer
- I called it a processing *model*
- It's a model because it's a simplified way of thinking about what's happening
  - It omits myriads of practical details that affect actual processors
  - We can still think about them in terms of the model
  - It is close enough to the truth that it lets us predict things correctly

> *"All models are wrong, but some are useful."*
> \- George E. P. Box

NTNU – Trondheim
Norwegian University of Science and Technology

The most abstract model



Question + Computer = Answer

If you have people who do the programming for you, this can be a sufficiently detailed view...

Breaking it down

- When *we* do the programming, some additional detail is necessary

Problem model	What we want to calculate: simplified representation of a real thing
Programming model	The operations we use to express the calculation: simplified representation of machine instructions
Processing model	Our expectations about what the machine will do: simplified representation of hardware
Actual computer	Constellation of assorted metals and plastics: the part you can kick

(This 4-layered view is adapted from "*Scalable Programming Models for Massively Multicore Processors*", M. D. McCool, Proceedings of the IEEE, Vol 96, Issue 5)

TDT4200 in context

- This part is what we've spent almost all of our time on

Problem model	What we want to calculate: simplified representation of a real thing
Programming model	**The operations we use to express the calculation: simplified representation of machine instructions**
Processing model	Our expectations about what the machine will do: simplified representation of hardware
Actual computer	Constellation of assorted metals and plastics: the part you can kick

  - It's a worthwhile topic, we can't write any programs otherwise
  - We need some ideas about the other 3 as well, in order to explain why the programs run as fast as they do

Other models:

Speedup & scaled speedup

- Amdahl’s and Gustafson’s laws are very abstract
  - They ignore the fact that hardly any program can split its parallel work into however many parts we want
  - They don’t precisely predict run times we can measure: in practice, it’s almost impossible to run the same program at *exactly* the same speed two times in a row
- They still model something useful
  - We get realistic estimates of whether or not program performance will improve if we buy more hardware to run it on

    (...so these are *performance models)*

Other models:
Hockney's communication model

- This one is pretty simplified too
  - It ignores the fact that message latency is affected by the communication library call, the operating system, the microcontroller in the network interface, the condition of the network cable, *etc. etc.*
  - It also ignores the fact that messages are sharing the network capacity with every other program that communicates via the same wires
- It still models something useful
  - We can tell whether program performance is constrained by the size of its messages or by how many they are

NTNU – Trondheim
Norwegian University of
Science and Technology

The inventory so far

- We've got performance models for how many processors to involve at a time
- We've got a performance model for how much time they'll spend talking to each other

- We don't have one for how well the processors perform while working on their local problem parts
  - We've just been recording it with a clock

Processor benchmarking

- This is a bit of a spectator sport
  - Hardware vendors compete for the highest numbers because it brings customers
  - Measurements are made with strictly regulated version numbers of strictly regulated benchmark programs under strictly regulated runtime conditions, so that results can be compared
  - Magazines, web sites, and private home enthusiasts publish tables of measurements, make comparisons, argue about the methods used to obtain them, *etc. etc.*
  - It's also an important part of the bidding and approval process when you're purchasing a machine with a specific performance target

Popular processor metrics

- Since olden times, people have compared “MIPS”
  - “Millions of Instructions Per Second”, ostensibly
  - Also known as “Meaningless Indicator of Processing Speed”, because different instructions (obviously) are not interchangeable
  - FLOPS (floating-point operations per second) are similarly popular, and slightly more homogeneous
- Throughout the clock race, people compared clock frequencies
  - It’s very easy to compare GHz (or MHz) if we assume that all program speeds are proportional to the clock rate
  - Of course, they aren’t *really…*
- For some time after 2005(ish), people have been counting cores
  - Regardless of how well their programs utilize them
- Recently, we’ve had to contend with different *types* of cores on the same chip
  - “Performance” vs. “efficiency” variants
  - Both of those are easy to count, too

The issue with benchmarks

- No matter which way you spin it, you're only *really* measuring the speed of the benchmark program
  - We try to make benchmarks that are representative for bigger classes of program types
  - That's very difficult, and not entirely accurate
- In order to estimate how fast *your* particular program can run on a given computer, it's helpful to analyze what kind of work the code does most of
  - That's where we are going with this

Data movement and operations

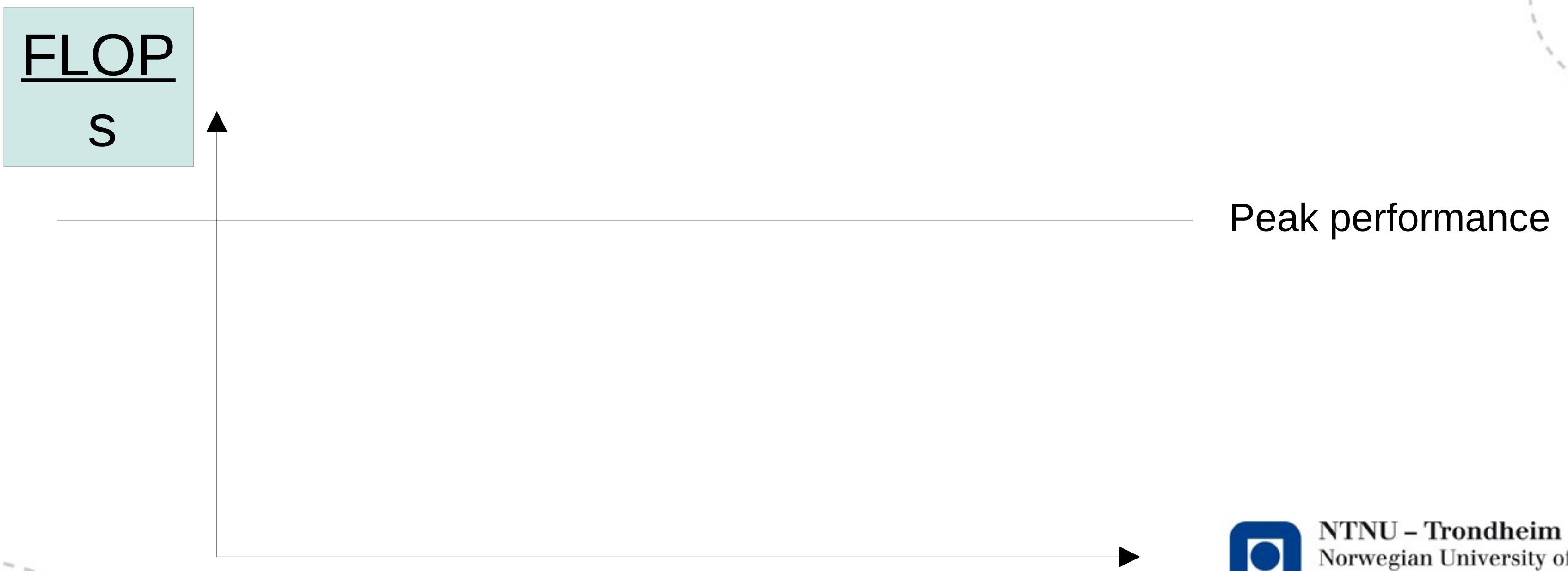
(again)

- As we have already noted several times, it can be useful to divide a program's work into
  - The parts that move numbers in and out of the processor (data movement)
  - The parts that combine numbers already in the processor (operations)
- Any given computer has some different costs for these
- We can choose what kind of operations to talk about, based on what the program is supposed to do
  - I'll talk in terms of FLOPS, because programs that do a lot of them tend to be performance-critical

    (...we have little use for performance-tuned text editors…)
  - There *are* performance-sensitive applications with different instruction mixes as well, you can adapt our discussion to those if you want/need to

We can draw a graph

- Let us make our performance metric the unit of the vertical axis
- Assuming that we just do a bunch of operations on registers (and don’t move any data), a computer has a peak computing rate



Data movement capacity

- The interconnect can maximally support shifting some number of bytes between CPU and memory each second
  - That's the *memory bandwidth*
  - Just like network bandwidth, in miniature
  - Measured in [bytes / s]

Operational intensity

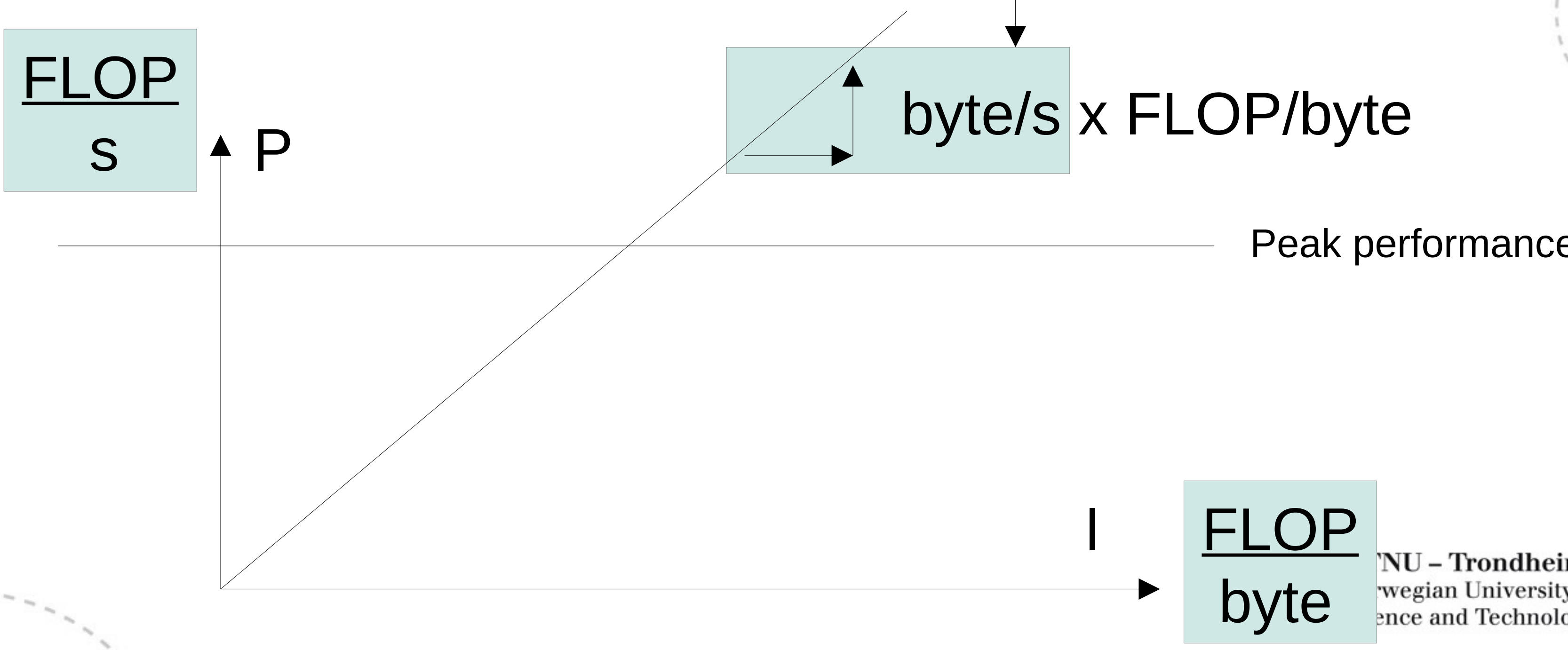
- Most instructions need some operands
  - We can sort out how many bytes those require
- All programs are composed from these instructions:
  - Read some number of bytes
  - Apply some operations to them
  - Write some number of bytes
- If we divide the number of ops by the number of bytes they are applied to, we get *operational intensity*
  - Measured in [operations / byte]
  - Also called *arithmetic intensity* when the program is full of arithmetic

The memory wall

- If the data transport is not fast enough to supply the processor with data for all the instructions in the program, we just have to wait for it to get there
- The **operational intensity** times the memory bandwidth becomes a performance figure
  - [byte / s] x **[FLOP / byte]** = [FLOP / s]
- This is as fast as the program can run because of the rate it can read and write at

Back to the graph

- If we make arithmetic intensity the unit of our x-axis, the machine's memory bandwidth gives the gradient of a straight line that relates them in our diagram:

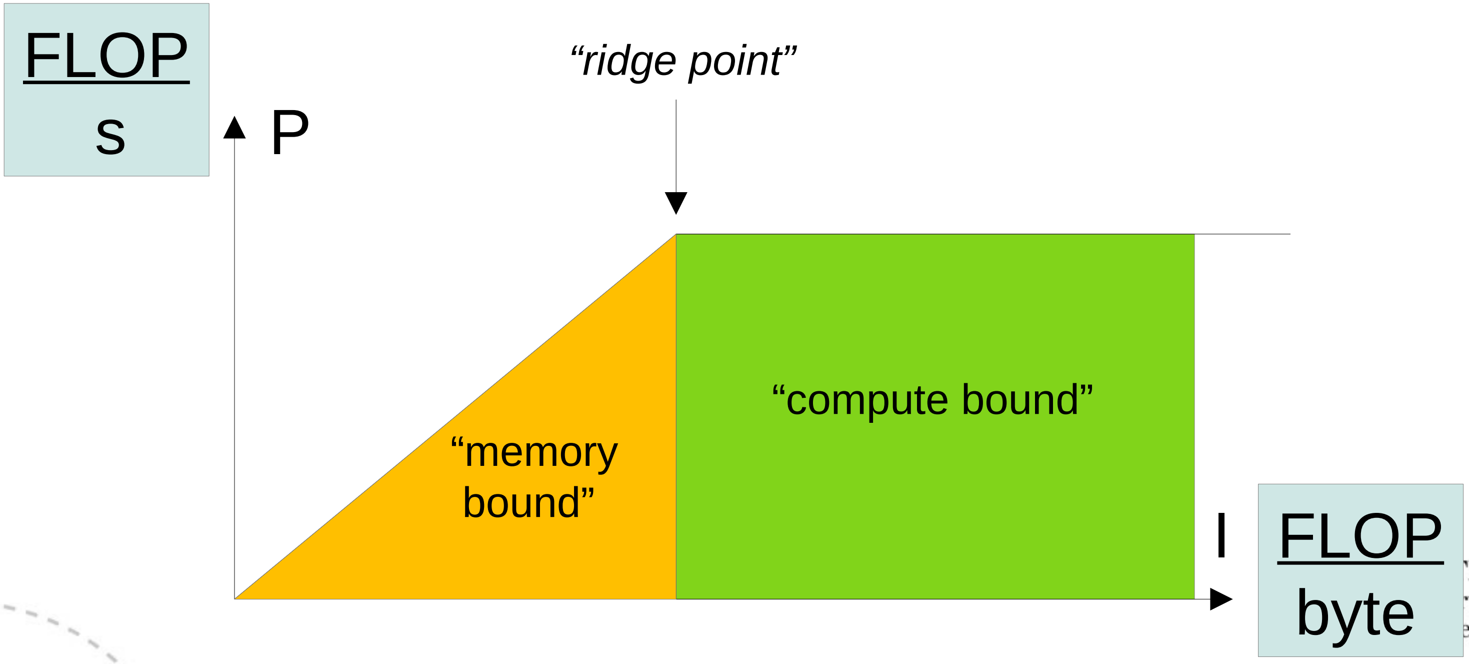


Roofline models

- The shape of this figure is determined by the maximal performance of a given computer
- It's a 'roofline' in the sense that performance can't exceed the computer's two maximum-capacities (memory bandwidth or peak operations rate)

We have two main regions

- Programs with intensity in the orange region will run at a speed capped by memory bandwidth
- Programs with intensity in the green region will run at a speed capped by the processor



How to find the arithmetic intensity?

- Read the code
  - A rough estimate can already be quite informative
- Here's the calculation from the advection example:

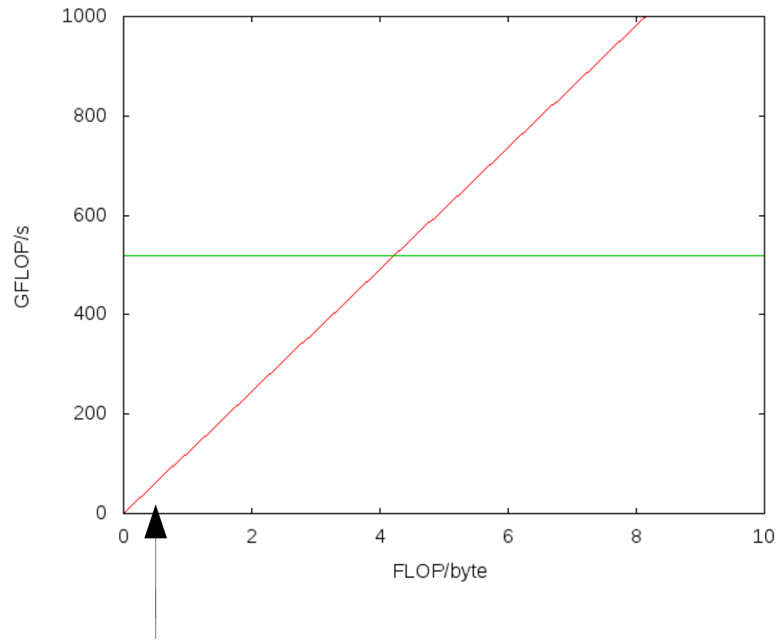
```
U_next(i,j) = 0.25 * (U(i-1,j) + U(i+1,j) + U(i,j-1) + U(i,j+1))
  - vy * (dt/(2.0*dx)) * (U(i+1,j) - U(i-1,j))
  - vx * (dt/(2.0*dx)) * (U(i,j+1) - U(i,j-1));
```

  - We've got 9 operands that are 8-byte floating point numbers, so that's 72 bytes

    (I only counted those that are liable to be loaded all the time, the others are likely to stay in cache after 1 initial load)
  - We've got 17 operations that are carried out every iteration
  - That's an intensity of approximately 0.236

What does this tell us?

- Here's a roofline chart I made for a 36-core Dell PE730 server:



The advection kernel is around here

→ The program will run at the speed of memory

How do we get the roofline?

- You can choose:
  - Theoretical numbers can be found in the data sheets of the hardware, but those are usually higher than you will ever see in practice
  - Empirical numbers can be found by running benchmarks that are known to specifically stress computing capability or memory, respectively
- I made the previous graph from timing
  - A *dgemm* multiplication with huge matrices (and optimized library)
  - A memory bandwidth benchmark called STREAM

It’s not an exact science

- We *could* have instrumented the program and obtained a more precise arithmetic intensity
  - It’s more work, though
  - As you can see, our approximation would have to be pretty bad before the result would change meaningfully
- We *could* have counted all the variables and constants in the expression
  - The intensity-number would have changed both value and meaning a little, I told you why I omitted them from this particular estimate
- There isn’t a single, 100% correct way to do it
  - If you want to put graphs like that in reports, documents, papers, *etc.,* just make sure that you include a description of how you got your numbers, and the reader will be able to tell what they mean