Optimization & Vectorization

J. Bikker & D. Alexandridis - April - June 2023 - Lecture 1: “Introduction”

Welcome!
Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling
Why?

Some problems require the supercomputer of the future.

- Anything that depends on Moore’s Law and time to become feasible.
Why?

*Games want to raise the bar:*

- More, better, faster. Also: be scalable.
Introduction

Why?

Some software needs to run on pretty weak hardware.

- Limited CPU, limited RAM (limited controls).
Why?

*Sometimes the cheapest / lowest power CPU is the best.*

- What is the lowest end CPU this will still run on? Can we go lower?
Perseverance, 2020

BRAINS
The Mars 2020 Rover's "Brains"

Unlike people and most animals, the rover's brains - its computer - are in its boxy body. The computer module is called the Rover Compute Element (RCE) - there are actually two identical RCEs in the body so there is always a spare "brain."

The Rover Compute Element interfaces with the engineering functions of the Perseverance rover over two networks which follow an aerospace industry standard designed especially for the high-reliability requirements of airplanes and spacecraft. In addition, the RCEs have a special purpose to direct interfaces with all of the rover's instruments for exchange of commands and science data.

Tech Specs

<table>
<thead>
<tr>
<th>Processor</th>
<th>Radiation-hardened central processor with PowerPC 750 Architecture, BAE RAD 750</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Operates at up to 200 megahertz speed, 10 times the speed in Mars rovers Spirit and Opportunity's computers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>2 gigabytes of flash memory (≈8 times as much as Spirit or Opportunity)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>256 megabytes of dynamic random access memory</td>
</tr>
<tr>
<td></td>
<td>256 kilobytes of electrically erasable programmable read-only memory</td>
</tr>
</tbody>
</table>
The PowerPC 740 and 750 (codename Arthur) were introduced in late 1997 as an evolutionary replacement for the PowerPC 603e. Enhancements included a faster 60x system bus (66 MHz), larger L1 caches (32 KB instruction and 32 KB data), a second integer unit, an enhanced floating point unit, and higher core frequency. The 750 had support for an optional 256, 512 or 1024 KB external unified L2 cache. The cache controller and cache tags are on-die. The cache was accessed via a dedicated 64-bit bus.

The 740 and 750 added dynamic branch prediction and a 64-entry branch target instruction cache (BTIC). Dynamic branch prediction uses the recorded outcome of a branch stored in a 512-entry by 2-bit branch history table (BHT) to predict its outcome. The BTIC caches the first two instructions at a branch target.

The 740/750 models had 6.35 million transistors and was initially manufactured by IBM and Motorola in an aluminium based fabrication process. The die measured 67 mm² at 0.26 μm and it reached speeds of up to 356 MHz while consuming 7.3 W.

In 1999, IBM fabricated versions in a 0.20 μm process with copper interconnects, which increased the frequency up to 500 MHz and decreased power consumption to 6 W and the die size to 40 mm².

The 740 slightly outperformed the Pentium II while consuming far less power and with a smaller die. The off-die L2 cache of the 750 increased performance by approximately 30% in most situations. The design was so successful that it quickly surpassed the PowerPC 604e in integer performance, causing a planned 604 successor to be scrapped.

The PowerPC 740 is completely pin compatible with the older 603, allowing upgrades to the PowerBook 1400, 2400, and even a prototype PowerBook 500/G3. The 750 with its L2 cache bus required more pins and thus a different package, a 360-pin ball grid array (BGA).

The PowerPC 750 was used in many computers from Apple, including the original iMac.
What is optimization?

Part of it is:

- INFOB3CC - Concurrency
- INFONW - Computerarchitectuur en netwerken
- INFOB3TC - Talen en compilers

And of course: any course that deals with improving existing algorithms.

Specific purpose of INFOMOV:

- To gain understanding of performance aspects of the hardware we use;
- To gain an intuition for what affects performance;
- To learn to apply a structured process to improve performance.
What is optimization?

**Think like a CPU**

- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD
What is optimization?

Work smarter, not harder: algorithm scalability

- **Big O**
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL, Boost: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory
Introduction

What is optimization?

Memory hierarchy: caches

- Cache architecture
- Cache lines
- Hits, misses and collisions
- Eviction policies
- Prefetching
- Cache-oblivious
- Data-centric programming

- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

- Big O
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory
What is optimization?

Don't assume, measure

- Profilers
- Interpreting profiling data
- Instrumentation
- Bottlenecks
- Steering optimization effort

- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

- Big O
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory

- Cache architecture
- Cache lines
- Hits, misses and collisions
- Eviction policies
- Prefetching
- Cache-oblivious
- Data-centric programming
What is optimization? – Project Management

Keeping code maintainable

- Pareto principle / 80-20 rule: roughly 80% of the effects are caused by 20% of the causes.
- 1% of the code takes 99% of the time.

“The curse of premature optimization”

- Optimization, rule 1: “Don’t do it”.
- Rule 2 (for experts only!), “Don’t do it yet”.

Optimization as a deliberate process

- Get predictable gains using a consistent approach.
What is optimization?

“Perceived Performance”

1. Wait for user input
2. Respond to user input \textit{as quickly as possible}
3. Execute requested operation.
At the end of this course:

*You will know how to speed up critical code by a factor 2.5x to 25x (and more).*

- You will be able to do this to virtually any program*.
- Your understanding of higher-level optimization approaches will increase.
- You will be able to apply these principles to new / alien hardware.
- You will have a more intimate relationship with your computer.

In other words:

We will talk a lot about the ‘C’ in O(N).

---

*disclaimer: ‘that has not been optimized by an expert’.
Today's Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling
Formalities

Lecturers

Jacco Bikker
i.bikker@uu.nl
Formalities

Lecturers

D. Alexandridis
d.alexandridis@uu.nl
Room 4.84 BBG
Course Layout

8 weeks + exam week:

- 2 lectures per week (for exceptions: see website)
- 1 or 2 guest lectures
- Lectures start at 09:00...
- Thursday working class starts at ~09:15, lecture at 10:00. 😊

Assessment:

- 2 assignments (25% each, individual or pairs);
- 1 final assignment (50%, individual or pairs);
- 1 final theory exam (individual).
Informalities

Media

Lectures will be live on campus

- But: also streamed live and recorded.

Working lectures are on campus only.

Slides will be distributed via the website.
Recordings will also be uploaded to the website.
Recordings can be found on Teams.
Prerequisites

C++

English

Hardware / software

You’ll need access to a computer with a CPU that supports SSE2 and OpenCL.

Obtaining VTune (Intel) is beneficial - VTune is free for students.

We will use Visual Studio 2019/22 (community edition).

Other tools will (also) be free.
Formalities

Literature

No book!
But that doesn’t mean you won’t be reading.

Main documents:


You are encouraged to do research into specific topics of interest yourself, and to report on this in class.
Audience

Any computer science student
(with a slight bias towards games)

Make sure you get as much as possible out of this course. This automatically includes a free pass.
Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling
Overview

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat step 6 and 7 until time runs out
Overview

Consistent Approach

(0.) Determine optimization requirements

- Target hardware (or range of hardware)
- Target performance
- Time available for optimization
- Constraints related to maintainability / portability
- ...

1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat steps 6 and 7 until time runs out

From here on, we will assume that:

- the code is ‘done’ (feature complete);
- a speed improvement is required;
- we have a finite amount of time for this.
Overview

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat steps 6 and 7 until time runs out
Overview

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
   ▪ caching, data-centric programming,
   ▪ removing superfluous functionality and precision,
   ▪ aligning data to cache lines, vectorization,
   ▪ checking compiler output, fixed point arithmetic,
   ▪ ...
8. Repeat steps 6 and 7 until time runs out
Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat steps 6 and 7 until time runs out
Overview

Assembler

In this course, we will not write assembler:

- It takes a pro to outperform the compiler
- You will be fighting the compiler
- You will have to redo the optimization for every target processor
- Maintainability will be zero.
“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

(Donald Knuth)
"A significant improvement in performance can often be achieved by solving only the actual problem and removing extraneous functionality." (Wikipedia)
“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.” (W.A. Wulff)
Quotes
"Dear Charles.

In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them (...).

One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

Therefore, one should attend INFOMOV and learn from it.

Love, Ada."
Today's Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling
Never Assume

Consistent Approach

(0.) Determine optimization requirements

1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out

Do you actually need to speed it up?
By how much?

Things to consider:

- You have a finite amount of time for this
- You don’t want to break anything
- You don’t want to reduce maintainability

Focus on ‘low hanging fruit’ – typically a small portion of the code.
Never Assume

Consistent Approach

1. **Profile: determine hotspots**
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out

---

Don't trust your intuition

- Not even when optimizing your own code.
- *Especially* not when you are proficient at optimizing.

Blind changes may reduce the performance of the code.

Needless to say: *use version control.*
Never Assume

Profiling

Measuring application performance

- Using external tools
- Using timers in the code

Measurements:

- How much time is spent were? (inclusive / exclusive, cycles, percentage)
- How often is each function called?
- Low level behavior: stalls / latencies, branch mispredictions, occupation, ...
- Performance over time: lag, spikes, stutter
What if the goal is to have a 10x larger army in your RTS?

Don’t just measure performance, measure *scalability*. 
More on profiling:

Next Week!
END of “Introduction”

next lecture: “Low Level”

After the break: Assignment 1 introduction.