Welcome!
Jonathan Adamczewski Retweeted

Mike Conley @mconley@mastodon.social @mike_conley · 13h
Are you interested in helping us to make Firefox mind-meltingly fast?

Come work with me!

careers.mozilla.org/position/gh/17...
Senior Firefox Performance Engineer

Apply for this job

Team:
Engineering

Locations:
Remote Canada, Toronto, Vancouver

Why join Mozilla engineering?

Engineering: Open Positions
Cloud Operations Engineer
Engineering Manager, Desktop Firefox Frontend
Engineering Manager, WebAssembly
Firefox Security Student Worker (Werkstudent)
Welcome!
Previously in INFOMOV...

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
Today’s Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement
What is the ‘cost’ of a multiply?

```c
starttimer();
float x = 0;
for( int i = 0; i < 1000000; i++ ) x *= y;
stoptimer();
```

- Actual measured operations:
  - timer operations;
  - initializing ‘x’ and ‘i’;
  - comparing ‘i’ to 1000000 (x 1000000);
  - increasing ‘i’ (x 1000000);
  - jump instruction to start of loop (x 1000000).

- Compiler outsmarts us!
  - No work at all unless we use x
  - x += 1000000 * y

Better solution:
- Create an arbitrary loop
- Measure time with and without the instruction we want to time
What is the ‘cost’ of a multiply?

```c
float x = 0, y = 0.1f;
unsigned int i = 0, j = 0x28929227;
for( int k = 0; k < ITERATIONS; k++ )
{
    // ensure we feed our line with fresh data
    x += y, y *= 1.01f;
    // integer operations to free up fp execution units
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
    // operation to be timed
    if (with) x *= y;
    // integer operations to free up fp execution units
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
}
dummy = x + (float)i;
```
Instruction Cost

x86 assembly in 5 minutes

Modern CPUs still run x86 machine code, based on Intel’s 1978 8086 processor. The original processor was 16-bit, and had 8 ‘general purpose’ 16-bit registers*:

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX (‘accumulator register’)</td>
<td>AH, AL (8-bit)</td>
</tr>
<tr>
<td>BX (‘base register’)</td>
<td>BH, BL</td>
</tr>
<tr>
<td>CX (‘counter register’)</td>
<td>CH, CL</td>
</tr>
<tr>
<td>DX (‘data register’)</td>
<td>DH, DL</td>
</tr>
<tr>
<td>BP (‘base pointer’)</td>
<td>EBP</td>
</tr>
<tr>
<td>SI (‘source index’)</td>
<td>EDI</td>
</tr>
<tr>
<td>DI (‘destination index’)</td>
<td>RDI</td>
</tr>
<tr>
<td>SP (‘stack pointer’)</td>
<td>ESP</td>
</tr>
</tbody>
</table>

Instruction Cost

x86 assembly in 5 minutes:

Typical assembler:

```
loop:
    mov eax, [0x1008FFA0]  // read from address into register
    shr eax, 5             // shift eax 5 bits to the right
    add eax, edx           // add registers, store in eax
    dec ecx                // decrement ecx
    jnz loop               // jump if not zero
    fld [esi]              // load from address [esi] onto FPU
    fld st0                // duplicate top float
    faddp                  // add top two values, push result
```

More on x86 assembler: [http://www.cs.virginia.edu/~evans/cs216/guides/x86.html](http://www.cs.virginia.edu/~evans/cs216/guides/x86.html)

What is the ‘cost’ of a multiply?

float x = 0, y = 0.1f;
unsigned int i = 0, j = 0x28929227;
for( int k = 0; k < ITERATIONS; k++ )
{
    // ...
    x += y, y *= 1.01f;
    // ...
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
    // ...
    if (with) x *= y;
    // ...
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
}
dummy = x + (float)i;

fldz
xor ecx, ecx
fld dword ptr ds:[405290h]
mov edx, 28929227h
fld dword ptr ds:[40528Ch]
push esi
mov esi, 0C350h

add ecx, edx
mov eax, 91D2A969h
xor edx, 17737352h
shr ecx, 1
mul eax, edx
fld st(1)
faddp st(3), st
mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1), st
xor edx, 17737352h
shr edx, 0Eh
dec esi
jne tobetimed<0>+1Fh

\[
\frac{246}{28763} = 50000
\]

infomov – lecture 2 – “low level”

11
Instruction Cost

What is the ‘cost’ of a multiply?

Observations:

- Compiler reorganizes code
- Compiler cleverly evades division
- Loop counter decreases
- Presence of integer instructions affects timing *(to the point where the mul is free)*

But also:

- It is really hard to measure the cost of a line of code.
Instruction Cost

What is the ‘cost’ of a single instruction?

Cost is highly dependent on the surrounding instructions, and many other factors. However, there is a ‘cost ranking’:

- **bit shifts**
- **simple arithmetic, logical operands**
- **multiplication**
- **division**
- sqrt
- sin, cos, tan, pow, exp

This ranking is generally true for any processor (including GPUs).
## Instruction Cost

### AMD K7

**1999**

Below is a table of instruction costs for the AMD K7 processor in 1999. The table lists various arithmetic instructions along with their operands, operations (Ops), latency, reciprocal throughput, execution unit, and notes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Ops</th>
<th>Latency</th>
<th>Reciprocal throughput</th>
<th>Execution unit</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, SUB</td>
<td>r, ri</td>
<td>1</td>
<td>1</td>
<td>1/3</td>
<td>ALU, AGU</td>
<td></td>
</tr>
<tr>
<td>ADD, SUB</td>
<td>r, m</td>
<td>1</td>
<td>1</td>
<td>1/2</td>
<td>ALU, AGU</td>
<td></td>
</tr>
<tr>
<td>ADD, SUB</td>
<td>m, r</td>
<td>1</td>
<td>7</td>
<td>2,5</td>
<td>ALU, AGU</td>
<td></td>
</tr>
<tr>
<td>ADC, SBB</td>
<td>r, ri</td>
<td>1</td>
<td>1</td>
<td>1/3</td>
<td>ALU</td>
<td></td>
</tr>
<tr>
<td>ADC, SBB</td>
<td>r, m</td>
<td>1</td>
<td>7</td>
<td>2,5</td>
<td>ALU</td>
<td></td>
</tr>
<tr>
<td>ADC, SBB</td>
<td>m, r</td>
<td>1</td>
<td>7</td>
<td>2,5</td>
<td>ALU</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td></td>
<td>1</td>
<td>36</td>
<td>12</td>
<td>FMUL</td>
<td></td>
</tr>
</tbody>
</table>

Math instructions include:

- `FSQRT`
- `FSIN`
- `FCOS`
- `FSIN COS`
- `FP TAN`
- `FP ATAN`
**Instruction Cost**

**AMD Jaguar 2013**

Note: Two micro-operations can execute simultaneously if they go to different execution pipes.
### Instruction Cost

#### Intel Silvermont 2014

Note: This is a low-power processor (ATOM class).

<table>
<thead>
<tr>
<th>Arithmetic instructions</th>
<th>Operands</th>
<th>pops</th>
<th>Unit</th>
<th>Latency</th>
<th>Reciprocal throughput</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD SUB</td>
<td>r, r/l</td>
<td>1</td>
<td>IP0/1</td>
<td>1</td>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td>ADD SUB</td>
<td>r, m</td>
<td>1</td>
<td>IP0/1, Mem</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ADD SUB</td>
<td>m, r/l</td>
<td>1</td>
<td>IP0/1, Mem</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ADC SBB</td>
<td>r, r/l</td>
<td>1</td>
<td>IP0/1</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ADC SBB</td>
<td>m, r/l</td>
<td>1</td>
<td>IP0/1</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>r, r/l</td>
<td>1</td>
<td>IP0/1</td>
<td>1/2</td>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>m, r/l</td>
<td>1</td>
<td>IP0/1</td>
<td>1/2</td>
<td>1/2</td>
<td></td>
</tr>
</tbody>
</table>

#### Math

<table>
<thead>
<tr>
<th>Math</th>
<th>Operands</th>
<th>pops</th>
<th>Unit</th>
<th>Latency</th>
<th>Reciprocal throughput</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSCALE</td>
<td></td>
<td>27</td>
<td>20</td>
<td>66</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>EXTRACT</td>
<td></td>
<td>15</td>
<td>13-40</td>
<td>60</td>
<td>13-40</td>
<td></td>
</tr>
<tr>
<td>FCOS</td>
<td></td>
<td>18</td>
<td>40-170</td>
<td>40</td>
<td>40-170</td>
<td></td>
</tr>
<tr>
<td>FCOS</td>
<td></td>
<td>110</td>
<td>40-170</td>
<td>10</td>
<td>40-170</td>
<td></td>
</tr>
<tr>
<td>P2XM1</td>
<td></td>
<td>34</td>
<td>80-140</td>
<td>65</td>
<td>80-140</td>
<td></td>
</tr>
<tr>
<td>FLY2X</td>
<td></td>
<td>61</td>
<td>154</td>
<td>101</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>FLY2X</td>
<td></td>
<td>101</td>
<td>45-200</td>
<td>63</td>
<td>45-200</td>
<td></td>
</tr>
<tr>
<td>FLY2X</td>
<td></td>
<td>63</td>
<td>85-190</td>
<td>66</td>
<td>85-190</td>
<td></td>
</tr>
</tbody>
</table>

Note: This is a low-power processor (ATOM class).
### Instruction Cost

#### Intel Skylake 2015

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand Size</th>
<th>Instruction</th>
<th>Operand Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD SUB r,r/l</td>
<td>1</td>
<td>ADD SUB r,m</td>
<td>2</td>
<td>ADD SUB m,r/l</td>
</tr>
<tr>
<td>ADC SBB r,r/l</td>
<td>1</td>
<td>ADC SBB r,m</td>
<td>2</td>
<td>ADC SBB m,r/l</td>
</tr>
<tr>
<td>CMP r,r/l</td>
<td>1</td>
<td>CMP m,r/l</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>INC DEC NEG NOT m</td>
<td>3</td>
<td>INC DEC NOT m</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>AAA</td>
<td>2</td>
<td>AAS</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**Math**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSCALE</td>
<td>27</td>
</tr>
<tr>
<td>FXTRACT</td>
<td>17</td>
</tr>
<tr>
<td>FSQRT</td>
<td>1</td>
</tr>
<tr>
<td>FSIN</td>
<td>53-105</td>
</tr>
<tr>
<td>FCOS</td>
<td>53-105</td>
</tr>
<tr>
<td>FSINCOS</td>
<td>55-120</td>
</tr>
<tr>
<td>F2XM1</td>
<td>16-90</td>
</tr>
<tr>
<td>FYL2X</td>
<td>40-100</td>
</tr>
<tr>
<td>FYL2XP1</td>
<td>56</td>
</tr>
<tr>
<td>FPATAN</td>
<td>40-112</td>
</tr>
<tr>
<td>FPATAN</td>
<td>30-160</td>
</tr>
</tbody>
</table>

**Note:** Not 64 bit
Instruction Cost

What is the ‘cost’ of a single instruction?

The cost of a single instruction depends on a number of factors:

- The arithmetic complexity (sqrt > add);
- Whether the operands are in register or memory;
- The size of the operand (16 / 64 bit is often slightly slower);
- Whether we need the answer immediately or not (latency);
- Whether we work on signed or unsigned integers (DIV/IDIV).

On top of that, certain instructions can be executed simultaneously.
Today’s Agenda:

- **The Cost of a Line of Code**
- **CPU Architecture: Instruction Pipeline**
- **Data Types and Their Cost**
- **Rules of Engagement**
CPU Instruction Pipeline

Instruction execution is typically divided in four phases:

1. **Fetch**: Get the instruction from RAM
2. **Decode**: The byte code is decoded
3. **Execute**: The instruction is executed
4. **Writeback**: The results are written to RAM/registers

\[
\text{CPI} = 4
\]
CPU Instruction Pipeline

For each of the stages, different parts of the CPU are active. To use its transistors more efficiently, a modern processor overlaps these phases in a *pipeline*.

At the same clock speed, we get four times the throughput (CPI = IPC = 1).
CPU Instruction Pipeline

Maximum clockspeed is determined by the most complex of the four stages. For higher clockspeeds, it is advantageous to increase the number of stages (thereby reducing the complexity of each individual stage).

Obviously, ‘execution’ of different instructions requires different functionality.

Superpipelining allows higher clockspeeds and thus higher throughput, but it also increases the latency of individual instructions.

### Stages

<table>
<thead>
<tr>
<th>Stages</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>PowerPC G4e</td>
</tr>
<tr>
<td>8</td>
<td>Cortex-A9</td>
</tr>
<tr>
<td>10</td>
<td>Athlon</td>
</tr>
<tr>
<td>12</td>
<td>Pentium Pro/II/III, Athlon 64</td>
</tr>
<tr>
<td>14</td>
<td>Core 2, Apple A7/A8</td>
</tr>
<tr>
<td>14/19</td>
<td>Core i2/i3 Sandy Bridge</td>
</tr>
<tr>
<td>16</td>
<td>PowerPC G5, Core i7 Nehalem</td>
</tr>
<tr>
<td>18</td>
<td>Bulldozer, Steamroller</td>
</tr>
<tr>
<td>20</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>31</td>
<td>Pentium 4E Prescott</td>
</tr>
</tbody>
</table>
Pipeline

CPU Instruction Pipeline

Different execution units for different (classes of) instructions:

Here, one execution unit handles floats;
one handles integer;
one handles memory operations.

Since the execution logic is typically the most complex part, we might just as well duplicate the other parts:
CPU Instruction Pipeline

This leads to the *superscalar* processor, which can execute multiple instructions in the same clock cycle, assuming not all instructions require the same execution logic.

IPC = 3 (or: ILP = 3)
CPU Instruction Pipeline

Using a pipeline has consequences. Consider the following situation:

```
# a = b * c;
d = a + 1;
```

Here, the second instruction needs the result of the first, which is available one clock tick too late. As a consequence, the pipeline stalls briefly.
CPU Instruction Pipeline

Using a pipeline has consequences. Consider the following situation:

```
a = b * c;
jump if a is not zero
```

In this scenario, a conditional jump makes it hard for the CPU to determine what to feed into the pipeline after the jump.
Pipeline

CPU Instruction Pipeline - Digest

For a more elaborate explanation of the pipeline, see this document: [http://www.lighterra.com/papers/modernmicroprocessors](http://www.lighterra.com/papers/modernmicroprocessors)

Or check this very detailed study of the Nehalem architecture:

For now:

- A compiler reorganizes code to prevent latencies
- Feeding mixed code provides the compiler with sufficient opportunities for shuffling
- Branching issues need to be prevented manually
Today’s Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement
Data Types

Data types in C++

- `int`
- `unsigned`

Size: 32 bit (4 bytes)

Access:

```
union { unsigned int u4; int s4; char s[4]; };  
unsigned char v = 100;  
s[1] = v;  
u4 = (u4 ^ (255 << 8)) | (v << 8);  
```

Altering sign bit of s4:  

```
(u4 ^ = 1 << 31;
```

```
Red = u4 & (255 << 16);  
Green = u4 & (255 << 8);  
Blue = u4 & 255;
```
Data Types in C++

- **float**

  - **Sign**: 1 bit; -1 or 1
  - **Exponent**: 8 bit; -127 ... 128
  - **Mantissa**: 23 bit; 0 ... 2^{23}-1

  Value: \( \text{sign} \times \text{mantissa} \times 2^{\text{exponent}} \)

  Size: 32 bit (4 bytes)

Exercise: write a function that replaces array \( a = \{ 0.5, 0.25, 0.125, 0.0625, \ldots \} \).
Data Types

Data types in C++

double 64 bit (8 bytes)
char, unsigned char 8 bit
short, unsigned short 16 bit
LONG 32 bit (same as int)
LONG LONG, __int64 64 bit
bool 8 bit (!)

Padding*

struct Test
{
    unsigned int u;
    double d;
    bool flag;
};

struct Test2
{
    bool flag;
};

// sizeof( Test ) is 8
// sizeof( Test2 ) is 16

* More on http://www.catb.org/esr/structure-packing
Data Types

Data types in C++  - Conversions

Explicit:

```cpp
float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);
```

Implicit:

```cpp
struct Color {
  unsigned char a, r, g, b;
};
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ ) {
  bitmap[i].r *= 0.5f;
  bitmap[i].g *= 0.5f;
  bitmap[i].b *= 0.5f;
}
```

```
// bitmap[i].r *= 0.5f;
movzx eax,byte ptr [ecx-1]
mov dword ptr [ebp-4],eax
fld dword ptr [ebp-4]
fnstcw word ptr [ebp-2]
movzx eax,word ptr [ebp-2]
or eax,0C00h
mov dword ptr [ebp-8],eax
fmul st,st (1)
fldcw word ptr [ebp-8]
fistp dword ptr [ebp-8]
movzx eax,byte ptr [ecx-1]
mov byte ptr [ecx-1],al
```
Data Types

Data types in C++ - Conversions

Explicit:

float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);

Avoiding conversion:

struct Color { unsigned char a, r, g, b; };
Color bitmap[640 * 480];
for (int i = 0; i < 640 * 480; i++)
{
    bitmap[i].r >>= 1;
    bitmap[i].g >>= 1;
    bitmap[i].b >>= 1;
}
Data Types

Data types in C++ - Conversions

Explicit:

```cpp
float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);
```

Avoiding conversion (2):

```cpp
struct Color {
    union {
        struct {
            unsigned char a, r, g, b;
        };
        int argb;
    };
}
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ )
{
    bitmap[i].argb = (bitmap[i].argb >> 1) & 0x7f7f7f7f;
}
```
Data Types

Data types in C++ - Free interpretation

Trick: Cheaper float comparison

```cpp
union { float v1; unsigned int u1; };
union { float v2; unsigned int u2; };
bool smaller = (v1 < v2);
bool smaller = (u1 < u2); // same result, if signs of v1 and v2 are equal.
```
Data Types

Data types in C++ - Rolling your own

HDR color storage

<table>
<thead>
<tr>
<th>exponent</th>
<th>red</th>
<th>green</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>24</td>
<td>23</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Storing a bit flag in a floating point value

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>mantissa</th>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>24</td>
<td>23</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement
Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 1: Avoid Costly Operations

- Replace multiplications by bitshifts, when possible
- Replace divisions by (reciprocal) multiplications
- Avoid sin, cos, sqrt
Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 2: Precalculate

- Reuse (partial) results
- Adapt previous results (interpolation, reprojection, ...)
- Loop hoisting
- Lookup tables
Common Opportunities in Low-level Optimization

RULE 3: Pick the Right Data Type

- Avoid byte, short, double
- Use each data type as a 32/64 bit container that can be used at will
- Avoid conversions, especially to/from float
- Blend integer and float computations
- Combine calculations on small data using larger data
Rules of Engagement

Common Opportunities in Low-level Optimization

RULE 4: Avoid Conditional Branches

- if, while, ?, MIN/MAX
- Try to split loops with conditional paths into multiple unconditional loops
- Use loop unrolling
- If all else fails: make conditional branches predictable
Common Opportunities in Low-level Optimization

RULE 5: Early Out

```c
char a[] = "abcdefghijklmnopqrstuvwxyz";
char c = 'p';
int position = -1;
for ( int t = 0; t < strlen( a ); t++ )
{
    if ( a[t] == c )
    {
        position = t;
    }
    else
    {
        if ( a[t] == c )
        {
            position = t;
            break;
        }
    }
}
```
Common Opportunities in Low-level Optimization

**RULE 6: Use the Power of Two**

- A multiplication / division by a power of two is a (cheap) bitshift
- A 2D array lookup is a multiplication too – make ‘width’ a power of 2
- Dividing a circle in 256 or 512 works just as well as 360 (but it’s faster)
- Bitmasking (for free modulo) requires powers of 2

Be fluent with powers of 2 (up to $2^{16}$);
learn to go back and forth for these: $2^9 = 512 = 2^9$.
Practice counting from 0..31 on one hand in binary.
Common Opportunities in Low-level Optimization

RULE 7: Do Things Simultaneously

- Use those cores
- An integer holds four bytes; use these for instruction level parallelism
- More on this later.
Common Opportunities in Low-level Optimization

1. Avoid Costly Operations
2. Precalculate
3. Pick the Right Data Type
4. Avoid Conditional Branches
5. Early Out
6. Use the Power of Two
7. Do Things Simultaneously
Today’s Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement
Practice

Get (from the website) project glassball.zip

Using low-level optimization, speed up this application.

1. Avoid Costly Operations
2. Precalculate
3. Pick the Right Data Type
4. Avoid Conditional Branches
5. Early Out
6. Use the Power of Two

Make sure functionality remains intact.

Target: a 10x speedup (this should be easy).
END of “Low Level”

next lecture: “caching (1)"