Branchless Programming in C++

Fedor G Pikus
Chief Scientist
PLAN

- Efficiency and performance
- Understanding the hardware and using it efficiently
  - Computing resources of a CPU
  - Pipelining
  - Branch prediction and hardware loop unrolling
- Conditional code vs efficiency
- Optimizing conditional code
- Branchless programming
WHAT CAN BRANCHLESS OPTIMIZATIONS DO?

f(bool b, unsigned long x, unsigned long& s) {if (b) s += x;}

- 130M calls/second
- Optimized: 400M calls/second

if (x[i] || y[i]) { ... }

- 150M evaluations/second
- Optimized: 570M evaluations/second
USE ALL OF THE CPU HARDWARE ALL THE TIME

- What determines performance?
- Optimal algorithm:
  - get the result with minimal work
- Efficient use of language:
  - do not do any unnecessary work
- Efficient use of hardware
  - use all available resources
  - at the same time
  - all the time
GLOSSARY OF HARDWARE

- New SSE4.2 Instructions
- Improved Lock Support
- Additional Caching Hierarchy
- Execution Units
- Memory Ordering & Execution
- Out-of-Order Scheduling & Retirement
- Instruction Decode & Microcode
- Instruction Fetch & L1 Cache
- L1 Data Cache
- L2 Cache & Interrupt Servicing
- Paging
- Branch Prediction
- Improved Loop, Streaming
- Deeper Buffers
- Simultaneous Multi-Threading
- Faster Virtualization
- Better Branch Prediction

- “Just works”
- Compiler takes care of it
- Needs care
- “Just works” but Good to know
unsigned long v1[N], v2[N];
unsigned long a = 0;
for (size_t i = 0; i < N; ++i)
{
    a += v1[i]*v2[i];
}
unsigned long v1[N], v2[N];
unsigned long a = 0;
for (size_t i = 0; i < N; ++i)
{
a += v1[i]*v2[i];
}
COMPUTING RESOURCES OF A CPU

register: i

memory: v1[i]
memory: v2[i]
COMPUTING RESOURCES OF A CPU

- register: i
- register: v1
- register: v2
- register: a1
- read: v1[i]
- memory: v1[i]
- memory: v2[i]
Computing Resources of a CPU

- Register: \( i \)
- Register: \( v_1 \)
- Register: \( v_2 \)
- Register: \( a_1 \)
- Memory: \( v_1[i] \)
- Memory: \( v_2[i] \)
- Read: \( v_2[i] \)
COMPUTING RESOURCES OF A CPU: USE ALL OF THE HARDWARE

register: i
register: v1
register: v2
register: a

multiply

memory: v1[i]
memory: v2[i]
A LOT OF CPU AREA IS DEDICATED TO COMPUTING. HAS TO BE GOOD FOR SOMETHING?

```c
unsigned long v1[N], v2[N];
unsigned long a1 = 0, a2 = 0;
for (size_t i = 0; i < N; ++i)
{
    a1 += v1[i]*v2[i];
    a2 += v1[i]+v2[i];
}
```
PROCESSORS CAN DO MULTIPLE OPERATIONS ON MULTIPLE REGISTERS AT ONCE

```
register: a1
register: v1
register: v2
... operations ...
register: a2
register: ...
```
A LOT OF CPU AREA IS DEDICATED TO COMPUTING. HAS TO BE GOOD FOR SOMETHING?

```c
unsigned long v1[N], v2[N];
unsigned long a1 = 0, a2 = 0;
for (size_t i = 0; i < N; ++i) {
    a1 += v1[i]*v2[i];
    a2 += v1[i]+v2[i];
    ...
}
```
USE MORE OF THE HARDWARE

- Using multiple compute units is easy when we have multiple independent computations
  - Life is rarely that good
- Usually results of one operation affect another operation
- Data dependency: \( a = (v_1 + v_2) \times (v_1 - v_2) \)
- Conditions, or branches: \( \text{if } (v > a) \ a = v; \)
  - Data-dependent code
Pipelining is the extension of the ability to execute multiple operations at once:

\[ a1 += (v1[i] + v2[i]) \times (v1[i] - v2[i]) \]

- \( s[i] = v1[i] + v2[i] \)
- \( d[i] = v1[i] - v2[i] \)

Data dependency: \( s[i] \times d[i] \)
PIPELINING: ANTIDOTE TO DATA DEPENDENCY

- Pipelining is the extension of the ability to execute operations at once:
  \[ a += (v1[i]+v2[i]) \times (v1[i]-v2[i]) \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s[i-1]:v1[i-1]+v2[i-1]</td>
<td>d[i-1]:v1[i-1]-v2[i-1]</td>
<td>s1[i-2]*d2[i-2]</td>
</tr>
<tr>
<td>s[i]:v1[i]+v2[i]</td>
<td>d[i]:v1[i]-v2[i]</td>
<td>s1[i-1]*d2[i-1]</td>
</tr>
<tr>
<td>s[i+1]:v1[i+1]+v2[i+1]</td>
<td>d[i+1]:v1[i+1]-v2[i+1]</td>
<td>s1[i]*d2[i]</td>
</tr>
</tbody>
</table>
• Using multiple compute units is easy when we have multiple independent computations
  - Usually results of one operation affect another operation
• Data dependency: \( a = (v_1 + v_2) \times (v_1 - v_2) \)
• Pipeline increases CPU utilization
• Multiple instruction streams run in parallel
  - Dependencies within each stream
  - No data dependencies between streams

USE MORE OF THE HARDWARE
BRANCHES: BANE OF THE PIPELINES

- Hard to pipeline code: \( a+ = (v1[i] > v2[i]) ? v1[i] : v2[i] \)

- Pipelining relies on a continuous stream of instructions
- Instructions are fetched, decoded, and executed
- Conditional jumps (branches) disrupt that order
- CPU must wait until it knows which instruction to fetch next
Not hard to pipeline code: \( a += (v1[i] > v2[i]) \? v1[i] : v2[i] \)

```
load:v1[i]
load:v2[i]
cmp[i]:v1[i] > v2[i]
v2[i] = v1[i] if true
a[i]: a += v2[i]
```

Conditional move
x86 cmove
BRANCHES: BANE OF THE PIPELINES

- Well-pipelined code: \( a += v1[i] + v2[i] \)
  - \( \text{load: } v1[i] \)
  - \( \text{load: } v2[i] \)
  - \( s[i]: v1[i] + v2[i] \)
  - \( a[i]: a += s[i] \)
  - \( \text{load: } v1[i+w] \)

  - \( \text{load: } v1[i+1] \)
  - \( \text{load: } v2[i+1] \)
  - \( s[i+1]: v1[i+1] + v2[i+1] \)
  - \( a[i+1]: a += s[i+1] \)
  - \( v1[i+2]: \)
  - \( v2[i+2]: \)
  - \( s[i+2]: \)
  - \( a[i+2]: \)

- Cannot run the pipeline for \( i+2 \) before checking that \( i+2 < N \)!
Branchless Computing

BRANCH PREDICTION: ANTIDOTE TO BRANCHES

• Well-pipedined code: \( a += v1[i] + v2[i] \)
• CPUs have branch predictors

Usually \( i < N \)
for (size_t i = 0; i < N; ++i) {
    a += v1[i]+v2[i];
}

- CPU immediately goes to the next iteration without waiting for i<N
  a += v1[1]+v2[1];
  a += v1[2]+v2[2];
  a += v1[3]+v2[3];
  ...
- Successive iterations are pipelined
- Hardware loop unrolling
LOOP UNROLLING – HOW?

- Machine code does not show any unrolling
  
  ```c
  for (size_t i = 0; i < N; ++i) {
      a += v1[i] + v2[i];
  }
  ```

- How can next stage of the pipeline run if registers are still in use?
- Register renaming: “rcx” does not mean “rcx”, CPUs have a lot more physical registers that are aliased to architecture register names like “eax” or “rcx”
- Result is hardware loop unrolling
  - Also out of order execution (data hazard)
BRANCHES: BANE OF THE PIPELINES

- Hard to pipeline code: 
  
  \[ a[i] = a + (v3[i]) ? (v1[i] + v2[i]) : (v1[i] \times v2[i]) \]

- Pipelining relies on a continuous stream of instructions.
- Instructions are fetched, decoded, and executed.
- Conditional jumps (branches) disrupt that order.
- CPU must wait until it knows which instruction to fetch next.

```
load:v1[i]...v3[i]
cmp[i]:v3[i]==0
jump if true
a[i]:a+=v1[i]+v2[i]
jump
a[i]:a+=v1[i]*v2[i]
...
```
BRANCH PREDICTION: ANTIDOTE TO BRANCHES

- Speculatively pipelined code: 

\[ a \text{ += } (v_3[i]) \ ? \ (v_1[i]+v_2[i]) : (v_1[i] \times v_2[i]) \]

```
load: v1[i]...v3[i]
cmp[i]: v3[i] == 0
jump if true
a[i]: a += v1[i]+v2[i]
jump
a[i]: a += v1[i] \times v2[i]
...
```

```
load: v1[i+1]...v3[i+1]
cmp[i+1]: v3[i+1] == 0
jump if true
a[i+1]: a += v1[i+1] \times v2[i+1]
```
Branchless Computing

**BRANCH PREDICTION: ANTIDOTE TO BRANCHES**

- Speculatively pipelined code:
  
  \[ a += (v3[i]) \ ? \ (v1[i]+v2[i]) : (v1[i]\times v2[i]) \]

- Performance critically depends on how effective the predictor is

```
load:v1[i]...v3[i]
cmp[i]:v3[i]==0
jump if true
a[i]:a+=v1[i]+v2[i]
jump
a[i]:a+=v1[i]\times v2[i]
...```

...
Well-pipelined code: \( a += v1[i] + v2[i] \)  
- CPUs have branch predictors  
- Branch predictors are associative caches, they remember the outcome of the conditional for the same place in the code  
- CPU assumes that the same branch will be taken (\( i < N \)) and proceeds to pipeline and evaluate instructions  
- Actual result of the conditional becomes known several cycles later  
- If the prediction was correct, nothing else needs to happen  
- If the prediction was wrong…
Branchless Computing

BRANCH MISPREDSICTIONS

- If branch prediction was wrong, several things need to happen:
  - All predicted computations are discarded or aborted
    - Pipeline flush
  - New computations have to be started
  - Any results of mispredicted computations have to be undone
    - Anything that cannot be undone cannot be done speculatively
if (p != NULL) *p = 1;  // p is rarely NULL
int v[N];
for (size_t i=0; i<N; ++i) {
    v[i] = i;            // Usually i<N
}

BRANCH MISPREDICTIONS AND ERRORS
if (p != NULL) *p = 1; // p is rarely NULL
int v[N];
for (size_t i=0; i<N; ++i) {
    v[i]=i; // Usually i<N
}

- Any errors are held until branch is evaluated
- Errors that do not actually happen must not be reported
- Memory writes must be held (destination may not be accessible)
BRANCH PREDICTION: ANTIDOTE TO BRANCHES

- Well-pipelined code: \( a += v1[i] + v2[i] \)

- Branch misprediction and pipeline flush at the end of the loop
- Branch predictor is effective – pipelining works – CPU utilization is good
v1 = ... some data ...;
v2 = ... some data ...;
v3[i] = 0;
//v3[i] = 1;
//v3[i] = rand();
for (size_t i = 0; i < N; ++i) {
    if (v3[i]) a1 += v1[i]+v2[i];
    else a2 += v1[i]*v2[i];
}
RESOURCES

- Google Benchmark:
  - https://github.com/google/benchmark

- Perf:
  - Usually part of Linux distribution
  - https://perf.wiki.kernel.org/index.php/Main_Page
  - Manual install involves compiling the kernel
BENCHMARK

- 01a
- 01b
- with perf
Branchless Computing

Branch Misprediction is Very Expensive

- $v_3[i] = 0$:
  - perf stat ./branch_predictions
  - 0.05% branch misses

- $v_3[i] = \text{rand}()$:
  - perf stat ./branch_predictions
  - 10% branch misses

- Optimizations to eliminate conditionals are usually invasive and may use more memory
- Branch predictors are quite complex
- Do not optimize until misprediction is confirmed by a profiler
BENCHMARK

- 01c
- with perf
BRANCH MISPREDICTION IS VERY EXPENSIVE

- Optimizations to eliminate conditionals usually are invasive and may use more memory
- Branch predictors are quite complex
  - Patterns in branch conditions are recognized
  - Differences in call stacks are detected
- Do not optimize until misprediction is confirmed by a profiler
WHAT IS A BRANCH?

if (x || y) do_it(); else dont_do_it();

• Programmer’s view:
  – if we always do it, branch is predictable

• Processor’s view:
  – if x is always true (or false), first branch is predictable
  – if y is always true (or false) whenever x is false, second branch is predictable
WHAT IS A BRANCH?

if (x || y) do_it(); else dont_do_it();

• Programmer’s view:
  – if we always do it, branch is predictable

• Processor’s view:
  – if x is always true (or false), first branch is predictable
  – if y is always true (or false) whenever x is false, second branch is predictable

• Root of the difference: Boolean expression evaluation is short-circuited
  – Evaluation must stop when the result is known
  – Important: if (*a || *b) … - b may be null whenever *a is true

• May be very expensive if the Boolean expression is complex, terms vary, but the overall result is predictable
BENCHMARK

- 02a
- with perf
OPTIMIZING FALSE BRANCH

if (x || y) do_it(); else dont_do_it();

- x may be true or false
- y may be true or false
- x || y is usually true
- Temporary variable:
  
  ```
  bool cond = x || y; if (cond) ...
  ```
  - Does not work at all:
  - compiler will get rid of it
  - it’s still two branches
OPTIMIZING FALSE BRANCH

if (x || y) do_it(); else dont_do_it();

- x may be true or false
- y may be true or false
- x || y is usually true
- Integer or bitwise arithmetic on bool:
  if (bool(x) + bool(y)) ... or if (bool(x) | bool(y)) ...
  - Works great unless the compiler “optimizes” operator + to ||
  - Some compilers do this (often for + or | but not both), some don’t
  - Profiling and/or examining assembly output is necessary
BENCHMARK

- 02b, 02c
- with perf
BRANCHES ARE THERE TO AVOID UNNECESSARY WORK

- Optimizing away branches almost always results in doing more work!
  \[
  \text{if} \ (x + y) \ldots
  \]
- Always evaluates \( x \) and \( y \)
- Always evaluates the sum
  \[
  \text{if} \ (x \ || \ y) \ldots
  \]
- Always evaluates \( x \), maybe \( y \)
- Does not evaluate \( || \) if \( x \) is true
- \( || \) is less work
Branchless Computing

BRANCHES ARE THERE TO AVOID UNNECESSARY WORK

- Optimizing away branches almost always results in doing more work!
- CPU usually has idle compute resources – can handle a bit of extra work
- Branch misprediction is very expensive
  - Predicted branch is just another instruction
- Tradeoff between the extra work vs the cost of the branch is usually impossible to predict – it must be measured
IF ONE BRANCH IS BETTER THAN TWO, THEN ZERO BRANCHES IS BETTER THAN ONE

- Branchless computing – eliminate branches completely, but how?
  \[ \text{sum} += \text{cond} \ ? \ \text{expr1} : \text{expr2}; \]

- Branchless implementation uses Booleans as integers
  \[ \text{term}[2] = \{ \text{expr2, expr1} \}; \]
  \[ \text{sum} += \text{term}[\text{bool}(\text{cond})]; \]

- Both expressions are evaluated

- Improves performance if:
  - extra computations are small
  - branch is poorly predicted
BENCHMARK

- 03a, b – branch is not predicted, optimization works
- 03c, d – branch is well-predicted, no optimization
Sometimes the compiler **will do** a branchless transformation for you
  - Often using “conditional move” instructions (they are not branches)

Compiler’s branchless optimization is usually better than yours

In particular, this is almost always branchless in reality:
```c
return cond ? x : y;
```

Never optimize such code preemptively

Optimize only if the profiler shows high misprediction rate

Optimizations depend on the compiler!
BENCHMARK

- 04c, d – optimization does not work with GCC
- with perf – no branch
Sometimes the compiler **will not do** a branchless transformation for you

This is almost always branchless in reality:

```c
return cond ? x : y;
```

But very similar code may not be

Never optimize such code preemptively

Optimize only if the profiler shows high misprediction rate
BENCHMARK

- 05a, b – optimization does work
- with perf – bad branch
ADVANCED OPTIMIZATION – ALWAYS MEASURE

• Sometimes branchless code is not really branchless
• Indirect function calls are similar to branches
  if (cond) f1(); else f2();
• Can be converted to branchless:
  funcptr f[2] = { &f2, &f1 };
  (f[cond])();
• This “optimization” almost never works
  – If f1() and/or f2() were inlined, it’s a spectacular pessimization
• Be careful – always measure
BENCHMARK

- 06a, b – optimization does not work
- with perf – bad branch either way
SUMMARY

- For best performance, use the hardware efficiently
- Use all of the hardware all the time (ideal goal)
- Processors can do many computations at once every cycle
- Limiting factor is usually availability of data
- Workaround is pipelining – running multiple instruction streams at once
- Limiting factor is conditional code – next instruction is data-dependent
- Workaround is branch prediction – guess the next instruction and go on
- Limiting factor is the ability to guess the future
- Workaround is writing unconditional code with data dependencies
LESSONS LEARNED

- Predicted branches are cheap
- Mispredicted branches are very expensive – pipeline flush
- Optimization – use fewer (or zero!) branches
- Always use profiler to detect and validate optimization locations
- Don’t fight with the compiler – sometimes it does the job for you
Illustrations by Evgenia Golant
Questions?

The Art of Writing Efficient Programs
An advanced programmer’s guide to efficient hardware utilization and compiler optimizations using C++ examples

Fedor G. Pikus

https://www.amazon.com/gp/mpc/A9QOPWSBTBFK4