Boosting Linux* Performance with GCC/GLIBC Latest Technologies

Intel SSG – OTC: SW Engineer: Victor Rodriguez

*Other names and brands may be claimed as the property of others.
Unused resources...
Let’s start with something simple: Adding two arrays

```c
#define MAX 1000000
int a[256], b[256], c[256];
int main () {
    int i, j;
    for (j=0; j<MAX; j++) {
        for (i=0; i<256; i++) {
            a[i] = b[i] + c[i];
        }
        return 0;
    }
}
```
Vectorization ... what is it?

![Vectorization Diagram](image-url)
Vectorization ...this is it?

```
+     +     +     +
+     +     +     +
```
Vectorization ... How to enable?

$ gcc -fopt-info-vec sanity.c -O2 -ftree-vectorize
$ gcc -fopt-info-vec sanity.c -O3

https://github.com/VictorRodriguez/autofdo_tutorial/blob/master/sort.c
Is it scalable?
Why don’t we extend the register more?
Intel® Advanced Vector Extensions (Intel®AVX), Intel® Advanced Vector Extensions 2 (Intel®AVX2), Intel® Advanced Vector Extensions 512 (Intel®AVX 512)

$ gcc -03 sanity.c -fopt-info-vec -mavx2 -o sanity

<table>
<thead>
<tr>
<th></th>
<th>511 -&gt; 256</th>
<th>255 -&gt; 128</th>
<th>127 - &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel AVX 512</td>
<td>Intel AVX2/Intel AVX</td>
<td>SSE</td>
<td></td>
</tr>
<tr>
<td>ZMM0</td>
<td>YMM0</td>
<td>XMM0</td>
<td></td>
</tr>
<tr>
<td>ZMM1</td>
<td>YMM1</td>
<td>XMM1</td>
<td></td>
</tr>
<tr>
<td>ZMM2</td>
<td>YMM2</td>
<td>XMM2</td>
<td></td>
</tr>
<tr>
<td>ZMM3</td>
<td>YMM3</td>
<td>XMM3</td>
<td></td>
</tr>
<tr>
<td>ZMM4</td>
<td>YMM4</td>
<td>XMM4</td>
<td></td>
</tr>
<tr>
<td>ZMM5</td>
<td>YMM5</td>
<td>XMM5</td>
<td></td>
</tr>
<tr>
<td>ZMM6</td>
<td>YMM6</td>
<td>XMM6</td>
<td></td>
</tr>
</tbody>
</table>
Intel® AVX technology advantage

Relative Performance, Higher Is Better

- With AVX2: 23.2 (23 X)
- With Vectorization: 15.9 (16 X)
- Without vectorization: 1.0
Let’s continue with another simple function...

Matrix Multiplication

```c
int i, j, k;
   for (i = 0; i < N; i++)
       {
       for (j = 0; j < N; j++)
           {
           for (k = 0; k < N; k++)
               C[i][j] += A[i][k]*B[k][j];
           }
       }
```
Fused Multiply and Add (FMA)

Floating-point multiply–add operation performed in one instruction

(2) _mm_fmadd_ss/sd

| Multiply and add the lowest element in the vectors |
| (res[0] = a[0] * b[0] + c[0]) |

```c
#include <immintrin.h>
#include <stdio.h>

int main() {
    __m128d veca = _mm_setr_pd(1.0, 2.0);
    __m128d vecb = _mm_setr_pd(5.0, 10.0);
    __m128d vecc = _mm_setr_pd(7.0, 14.0);
    __m128d result = _mm_fmadd_pd(veca, vecb, vecc);
    double* res = (double*)&result;
    printf("%lf %lf\n", res[0], res[1]);
    return 0;
}
```

# gcc fma.c -mfma

$ a \leftarrow a + (b \times c)$
• Assuming Port 0 FMA has a latency of 4 cycles, on previous example FMA is 2X faster than execute instructions separately.
• FMA instruction can speed up and improve the accuracy of many FP calculations. Intel® architecture (code name Haswell) implements FMA instructions.

OLD

\[
\text{loop: }
\begin{align*}
\text{vmulps ymm4, ymm0, ymm2} & \quad / A \times C2 \\
\text{vpaddps ymm0, ymm1, ymm4} & \\
\text{dec eax} & \\
\text{jnz loop} & 
\end{align*}
\]

4 cycles

NEW

\[
\text{loop: }
\begin{align*}
\text{vfmadd132ps ymm0, ymm1, ymm2} & \quad / C1 + A \times C2 \\
\text{dec eax} & \\
\text{jnz loop} & 
\end{align*}
\]

4 cycles

2X FASTER

OLD

NEW
Matrix Multiplication is a core building block of machine learning and big data.
Great !!! How do I implement this on Linux* distributions?


*Other names and brands may be claimed as the property of others.
How do I apply that to Linux*?

- Multiple files
- One file
- GLIBC
- FMV

*Other names and brands may be claimed as the property of others.
GLIBC solution in 2.26 !!!

x86: Set dl_platform and dl_hwcap from CPU features

Transparent linker to determine platform and build an array of hardware capability names

Added to search path when loading shared object.

Thanks to H.J. Lu !!!

https://upload.wikimedia.org/wikipedia/commons/e/ee/Rack_railway_turnout_%28SPB%29.JPG
static double a[4] = {1,2,5,6};
static double b[4] = {3,4,7,8};
static double c[4] = {17,20,57,68};
int N = 2;
double res;
for (int i = 0; i < N; i++)
    for (int j=0; j<N; j++)
        res = cblas_ddot(N,&a[N*i],1,&b[j],2);

• openat(AT_FDCWD, "/usr/lib64/haswell/libopenblas.so.0", O_RDONLY|O_CLOEXEC) = 3
• read(3</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>,
• fstat(3</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>,
• mmap(</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>)
• close(3</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>) = 0

---

STRACE Without haswell lib

• openat(AT_FDCWD, "/usr/lib64/haswell/libopenblas.so.0", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
• openat(AT_FDCWD, "/usr/lib64/tls/haswell/libopenblas.so.0", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
• openat(AT_FDCWD, "/usr/lib64/tls/libopenblas.so.0", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
• openat(AT_FDCWD, "/usr/lib64/haswell/libopenblas.so.0", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
• openat(AT_FDCWD, "/usr/lib64/libopenblas.so.0", O_RDONLY|O_CLOEXEC) = 3</usr/lib64/libopenblas_sandybridgep-r0.2.20.so>
• read(3</usr/lib64/libopenblas_sandybridgep-r0.2.20.so>,
• fstat(3</usr/lib64/libopenblas_sandybridgep-r0.2.20.so>,
• mmap(</usr/lib64/libopenblas_sandybridgep-r0.2.20.so>)
• close(3</usr/lib64/libopenblas_sandybridgep-r0.2.20.so>) = 0

---

STRACE With haswell lib

• openat(AT_FDCWD, "/usr/lib64/haswell/libopenblas.so.0", O_RDONLY|O_CLOEXEC) = 3</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>
• read(3</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>
• fstat(3</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>
• Mmap(3</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>, 0)
• close(3</usr/lib64/haswell/libopenblas_haswellp-r0.2.20.so>) = 0
Intel® AVX2 (vectors)

/usr/lib64/haswell/avx512_1/libfftw3f.so.3.5.6
- vmovsldup %zmm3,%zmm3
- vsubps %zmm0,%zmm8,%zmm3

/usr/lib64/haswell/libopencv_highgui.so.3.3.0
- vaddpd (%rdi),%xmm0,%xmm0

/usr/lib64/haswell/libbf_render.so
- vsubps %ymm1,%ymm7,%ymm1

FMA (throughput optimization)

/usr/lib64/haswell/avx512_1/libfftw3f.so.3.5.6
- vfmadddsub132ps %zmm3,%zmm4,%zmm7
- Vfmadddsub231ps %zmm2,%zmm8,%zmm3
- vfmadddsub132ss %xmm17,%xmm3,%xmm14

/usr/lib64/haswell/libopencv_highgui.so.3.3.0
- vfmaddd132sd -0xd0(%rbp),%xmm4,%xmm0
- vfmaddd132sd -0xd8(%rbp),%xmm6,%xmm0
Problem?

How many libraries should I deploy?

Why don’t we put everything in the same box?

Intel®AVX2

Intel®ATOM

Intel®AVX

SSE

avx2

avx

sse
First ideas came a long time ago
...But it was just for C++

```c
__attribute__((target("sse4.2")))
int foo(){
    // foo version for SSE4.2
    return 1;
}
__attribute__((target("arch=atom")))
int foo(){
    // foo version for the Intel Atom processor
    return 2;
}

int main() {
    int (*p)() = &foo;
    assert((*p)() == foo());
    return 0;
}
```

The target() directives will compile the functions for instruction-set extensions (e.g. sse4.2) or for specific architectures (e.g. arch=atom).

http://lwn.net/Articles/691932/
• Here, for each function, the developer needed to create specific functions and code for each target.

• That would have required extra overhead in the code; increasing the number of LOC in a program for FMV makes it more **clunky** to manage and maintain.
Function Multiversioning (FMV) since GCC

#define MAX 1000000
int a[256], b[256], c[256];

__attribute__((target_clones("avx2","arch=atom","default")))
void foo(){
    int i,x;

    for (x=0; x<MAX; x++){
        for (i=0; i<256; i++){
            a[i] = b[i] + c[i];
        }
    }
}

int main() {
    foo();
    return 0;
}

Thanks to !!!
Evgeny Stupachenko
Objdump?

- No Vectorization
- SSE
- Intel®AVX
- Intel®AVX2

You can read more at LWN article: https://lwn.net/Articles/691932/

a[i] = b[i] + c[i];
add 0x601060(%rax),%edx

a[i] = b[i] + c[i];
padd 0x602050(%rax),%xmm0

a[i] = b[i] + c[i];
vpadd 0x602050(%rax),%xmm0,%xmm0

a[i] = b[i] + c[i];
vpadd (%r8,%rax,1),%ymm0,%ymm0
CPUID selection (explain what CPUID is)

- In GCC* 4.8, FMV had a dispatch priority rather than a CPUID selection.
- Function versions with more advanced features got higher priority.
- For example, a version targeted for Intel® AVX2 would have a higher dispatch priority than a version targeted for SSE2.

- In GCC 6, the resolver checks the CPUID and then calls the corresponding function. It does this once per binary execution.
- With multiple calls to the FMV function, only the first call will execute the CPUID comparison; the subsequent calls will find the required version by a pointer.
- This technique is already used for almost all glibc functions <memcpy()>

*Other names and brands may be claimed as the property of others.
In case you don’t have time…
We do it for you !!!

https://github.com/clearlinux/make-fmv-patch
How these technologies help real applications?
Data analytics

https://upload.wikimedia.org/wikipedia/commons/9/9b/Social_Network_Analysis_Visualization.png
Four main languages for Analytics, Data Mining, Data Science

By Gregory Piatetsky at kdnuggets.com

*Other names and brands may be claimed as the property of others.
R language

v1 <- matrix(data=c(1,5,2,6),nrow=2,ncol=2)
v2 <- matrix(data=c(3,7,4,8),nrow=2,ncol=2)
v1 %*% v2

Answer

[,1] [,2]
[1,] 17 20
[2,] 57 68

Spec file -> https://github.com/clearlinux-pkgs/R
Results

pts/rbenchmark relative performance (Higher is Better)

Popular Linux distro
1 X

Clear Linux* OS for Intel® Architecture without /haswell/libs
1.1 X

Clear Linux OS for Intel Architecture with /haswell/libs
1.6 X

*Other names and brands may be claimed as the property of others.
Machine Learning

https://upload.wikimedia.org/wikipedia/commons/1/17/ArtificialFictionBrain.png
Scikit-learn

Machine Learning in Python*

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Most Popular Programming Languages For Machine Learning And Data Science by fossbytes

*Other names and brands may be claimed as the property of others.
Results

**pts/scikit-learn relative Performance (Higher is Better)**

- **Popular Linux distro**: 1 X
- **Clear Linux* OS for Intel® Architecture without /haswell/libs**: ~6X
- **Clear Linux OS for Intel Architecture with /haswell/libs**: ~7X

*Other names and brands may be claimed as the property of others.*
Easy to use as:

```bash
# docker run -it clearlinux/machine-learning
```

More info at clearlinux.org
Disclaimers

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.

Intel, the Intel logo and others are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

© 2017 Intel Corporation.
Magic, it must be remember, is an art which demands collaboration between the artist and the public

E. M. Butler

The Myth of the Magus

1948
Back up:

FMV Objdump sections:

<main>:
<get_available_features>:
<__cpu_indicator_init>:
<deregister_tm_clones>:
<register_tm_clones>:
<__do_global_dtors_aux>:
<frame_dummy>:
<foo>:
<foo.avx.0>:
<foo.avx2.1>:
<foo.arch_atom.2>:
<foo.resolver>:
<__libc_csu_init>:
<__libc_csu_fini>:
<fini>:
- Westmere
- SandyBridge
- IvyBridge
- Haswell
- Broadwell
- Intel® Xeon Phi™ Future Xeon processors

- AVX
- AVX2
- AVX-512