Improving Linux Performance with GCC latest technologies
Agenda

• Unused resources
• AVX technology
• Function Multi Versioning
• Auto FDO and Profiling
Unused resources....

https://i.ytimg.com/vi/X0ymEx64RLw/maxresdefault.jpg
#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?

e.g. 3 x 32-bit unused integers

A[1] + not used + not used + not used

B[1] + not used + not used + not used

C[1] + not used + not used + not used
Vectorization …this is it?
Vectorization ... How to enable?

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

<table>
<thead>
<tr>
<th>Example</th>
<th>GCC flags</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>sanity.c</td>
<td>gcc sanity.c -o sanity</td>
<td>603 ms</td>
</tr>
<tr>
<td>sanity.c</td>
<td>gcc -O3 sanity.c -o sanity</td>
<td>38 ms</td>
</tr>
</tbody>
</table>

Why don’t we extend the register more?
$ gcc -O3 sanity.c -fopt-info-vec -mavx2 -o sanity

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

Execution time of array addition (Lower is better)

- AVX2: 26 ms
- Vectorization: 38 ms
- None: 603 ms

Time (ms)
Intel AVX-512 will be first implemented in the future Intel® Xeon Phi™ processor and coprocessor known by the code name Knights Landing, and will also be supported by some future Xeon processors scheduled to be introduced after Knights Landing.
How do I apply that to Linux?

- Multiple Libraries
  - GLIBC
- One Libraries
  - FMV

```c
/* Ideally this should simply be __builtin_cpu_supports("avx2") */
/* However that does not check for OSXSAVE, XSTATE_SSE, XSTATE_YMM */
/* Origin gcc/testsuite/gcc.target/i386/avx2-check.h */
/* Origin gcc/testsuite/gcc.target/i386/avx-os-support.h */
+#ifdef __x86_64__
  unsigned int eax, ebx, ecx, edx;
+
+  if ((__get_cpuid (1, &eax, &ebx, &ecx, &edx)) {
+      if (__glibc_unlikely (GLRO(dl_debug_mask) & DL_DEBUG_LIBS))
+        _dl_debug_printf (" got cpuid\n");
+  } else {
+    if (__glibc_unlikely (GLRO(dl_debug_mask) & DL_DEBUG_LIBS))
+      _dl_debug_printf (" failed to get cpuid\n");
+    return 0;
+  }
```
/* Add 'avx2' capability on x86_64 */

```
+ if (__avx2_available())
+ {
+     if (__glibc_unlikely (GLRO(dl_debug_mask) & DL_DEBUG_LIBS))
+         _dl_debug_printf(" adding avx2 cap support\n");
+     temp[m].str = "avx2";
+     temp[m].len = 4;
+     ++m;
+ }
+ else {
+     if (__glibc_unlikely (GLRO(dl_debug_mask) & DL_DEBUG_LIBS))
+         _dl_debug_printf(" not adding avx2 cap support\n");
+     --cnt;
+ }
```
Make it work in `ldconfig`

```c
- glibc-2.21.orig/elf/ldconfig.c
+ glibc-2.21/elf/ldconfig.c
@@ -1297,6 +1297,7 @@ main (int argc, char **argv)
       under which TLS support was optional. The entry is no longer needed, but
       must remain for compatibility. */
       hwcap_extra[63 - _DL_FIRST_EXTRA] = "tls";
+ hwcap_extra[62 - _DL_FIRST_EXTRA] = "avx2";
```
In spec files:

```bash
%build
export AR=gcc-ar
export RANLIB=gcc-ranlib
export CFLAGS="$CFLAGS -ffito -fno-semantic-interposition -O3 "
export FFLAGS="$CFLAGS -ffito -fno-semantic-interposition -O3 -fno-f2c "
export CXXFLAGS="$CXXFLAGS -ffito -fno-semantic-interposition -O3 "

sed -i -e "s/\-O2/\-O3/g" Makefile*

pushd ..
  cp -a OpenBLAS-%{version} openblas-noavx
  cp -a OpenBLAS-%{version} openblas-avx2

pushd openblas-noavx
make TARGET=SANDYBRIDGE F_COMPILER=GFORTRAN SHARED=1 DYNAMIC_THREADS=1 NUM_THREADS=128 %{?_smp_mflags}
popd
export CFLAGS="$CFLAGS -march=haswell "
export FFLAGS="$FFLAGS -march=haswell -O3 "
pushd openblas-avx2
make TARGET=HASWELL F_COMPILER=GFORTRAN SHARED=1 DYNAMIC_THREADS=1 USE_OPENMP=1 NUM_THREADS=128 %{?_smp_mflags}
popd
```
%install

rm -rf %{buildroot}
export AR=gcc-ar
export RANLIB=gcc-ranlib
export CFLAGS="$CFLAGS -flto -ffunction-sections -fno-semantic-interposition -O3 "
export CXXFLAGS="$CXXFLAGS -flto -ffunction-sections -fno-semantic-interposition -O3 "

pushd ..

pushd openblas-noavx
make install DESTDIR=%{buildroot} PREFIX=/usr OPENBLAS_LIBRARY_DIR=/usr/lib64
popd

export CFLAGS="$CFLAGS -march=haswell "
pushd openblas-avx2
make install DESTDIR=%{buildroot} PREFIX=/usr OPENBLAS_LIBRARY_DIR=/usr/lib64/avx2
popd

popd
%files
%defattr(-,root,root,-)
/usr/include/cblas.h
/usr/include/f77blas.h
/usr/include/blas.h
/usr/include/lapacke.h
/usr/include/lapacke_config.h
/usr/include/lapacke_mangling.h
/usr/include/lapacke_utils.h
/usr/include/openblas_config.h
/usr/lib64/avx2/libopenblas.a
/usr/lib64/avx2/libopenblas.so
/usr/lib64/avx2/libopenblas.so.0
/usr/lib64/avx2/libopenblas_haswellp-r0.2.16.a
/usr/lib64/avx2/libopenblas_haswellp-r0.2.16.so
/usr/lib64/libopenblas.a
/usr/lib64/libopenblas.so
/usr/lib64/libopenblas.so.0
/usr/lib64/libopenblas_sandybridgep-r0.2.16.a
/usr/lib64/libopenblas_sandybridgep-r0.2.16.so
/usr/lib64/avx2/cmake/openblas/OpenBLASConfig.cmake
/usr/lib64/cmake/openblas/OpenBLASConfig.cmake
/usr/lib64/avx2/cmake/openblas/OpenBLASConfigVersion.cmake
/usr/lib64/cmake/openblas/OpenBLASConfigVersion.cmake
Real case : example 1

%configure --disable-static
   --without-x
   --with-system-zlib
   --with-system-bzlib
   --with-system-pcre
   --with-system-xz
   --enable-BLAS-shlib
   --enable-R-shlib
   --with-blas="-lopenblas"
   --with-cairo --enable-lto

R language
Open BLAS
Real case: example 1

**R Benchmark**
Phoronix Test Suite v6.2.0

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Seconds, Less Is Better</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Linux 6430</td>
<td>0.2062</td>
</tr>
<tr>
<td>Ubuntu 15.10</td>
<td>0.7691</td>
</tr>
<tr>
<td>Ubuntu 16.04 20160223</td>
<td>0.7755</td>
</tr>
<tr>
<td>Fedora 23</td>
<td>0.7922</td>
</tr>
</tbody>
</table>

1. Clear Linux 6430: R scripting front-end version 3.2.3 (2015-12-10)
2. Ubuntu 15.10: R scripting front-end version 3.2.2 (2015-09-14)
3. Ubuntu 16.04 20160223: R scripting front-end version 3.2.3 (2015-12-10)
4. Fedora 23: R scripting front-end version 3.2.3 (2015-12-10)
Problem?

How many binaries should I deploy?

- AVX2
- AMD
- AVX
- SSE
Why don’t we put everything in the same box

- avx2
- avx
- sse
In the old times and only in C++ (before GCC 6)

```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).
• 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.
• Fortunately, GCC 6 solves this problem: it supports FMV in both C and C++ code with a single attribute to define the minimum set of architectures to support.

• This makes it easier to develop Linux applications that can take advantage of enhanced instructions, without the overhead of replicating functions for each target.

Thanks to !!!
Evgeny Stupachenko
Function Multiversioning (FMV)

```c
#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;
}
```
We will build that with:

$ gcc -O3 sanity.c -o sanity

And you can watch the magic with objdump like this:

gcc -O3 sanity.c -g -c

objdump -S sanity.o
No AVX code (Atom):

```
add %eax, %edx
```

AVX:

```
vpadd 0x0(%rax), %xmm0, %xmm0
```

AVX2:

```
vpadd (%r9, %rax, 1), %ymm0, %ymm0
```
CPUID selection

• 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 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.
• So when there are 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. For example, glibc has memcpy() optimized for each architecture, so when it is called, glibc will call the proper optimized memcpy().
## Results

<table>
<thead>
<tr>
<th>GCC flags</th>
<th>Haswell</th>
<th>Skylake</th>
<th>Broadwell</th>
<th>Xeon</th>
<th>Atom</th>
<th>Ivy Bridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>603</td>
<td>645</td>
<td>580</td>
<td>1413</td>
<td>2369</td>
<td>517</td>
</tr>
<tr>
<td>-03</td>
<td>38</td>
<td>44</td>
<td>37</td>
<td>107</td>
<td>96</td>
<td>60</td>
</tr>
<tr>
<td>-03 -mavx</td>
<td>26</td>
<td>32</td>
<td>26</td>
<td>73</td>
<td>SIGILL</td>
<td>45</td>
</tr>
<tr>
<td>-03 -mavx2</td>
<td>26</td>
<td>32</td>
<td>26</td>
<td>73</td>
<td>SIGILL</td>
<td>SIGILL</td>
</tr>
<tr>
<td>-03 (with FMV)</td>
<td>26</td>
<td>32</td>
<td>26</td>
<td>73</td>
<td>96</td>
<td>45</td>
</tr>
</tbody>
</table>
Profiling…
Helping the compiler find the most efficient path…
Profiling.. Learning on the road

Invasive Profiling: PGO
- Advantages: Very precise
- Disadvantages: High overhead

Non-Invasive Profiling: AutoFDO
- Advantages: Small overhead
- Disadvantages: not precise: statistical data
MariaDB + PGO

![Bar chart showing response times for read/write and read-only operations on a ram disk, with lower times indicating better performance. Series 2 consistently outperforms Series 1 in both read/write and read-only operations.](image-url)
MariaDB + PGO

![Bar chart showing performance comparison between PGO and baseline for "read/write only (ram disk)" and "read only (ram disk)" scenarios. The chart indicates higher transaction per second for PGO in both cases.](image-url)
MariaDB + PGO + Open Stack

Average time (sec) to create 100 users (running 1000 times), lower is better
Phase 1: Generate the profile file

AutoFDO needs a perf.data file that captures the BR_INST_RETIRED:TAKEN event in the processor. This event will vary for every architecture, so we are going to use ocperf, which is part of the pmu-tools project, which wraps all the information required for perf to generate the perf.data correctly for any Intel architecture. The user is free to use this tool or just the perf tool.

```bash
# ocperf.py record -b -e br_inst_retired.near_taken:pp -- ./sort
Bubble sorting array of 30000 elements
3731 ms
[ perf record: Woken up 7 times to write data ]
[ perf record: Captured and wrote 1.580 MB perf.data (3902 samples) ]
```

After this a standalone tool is used to convert the perf.data file into gcov format. This tool is create_gcov from autofdo set of tools:

```bash
# create_gcov --binary=./sort --profile=perf.data --gcov=sort.gcov --gcov_version=1
```

Phase 2: Use profile to optimize binary

The following info is read from the profile gcov file (in our case sort.gcov):

- Function names and file names.
- Source level profile, which is a mapping from inline stack to its sample counts.
- Module profile: Module to aux-modules mapping

In order to read the profile file we need to rebuild the source:

```bash
# gcc -O3 -fauto-profile=sort.gcov sort.c -o sort_autofdo
```

After this we have the binary sort_autofdo, the which we can run to test:
AWK and AutoFDO?

AWK benchmark (lower is better)

With AutoFDO
Without AutoFDO

https://gcc.gnu.org/wiki/AutoFDO/Tutorial

Thanks to !!!
Andi Kleen
Bugs .. Not everything was pretty

- LTO
- AutoFDO
- FMV
- Macros
Unused resources....

https://i.ytimg.com/vi/X0ymEx64RLw/maxresdefault.jpg
Thank you