Hands-on: Vectorization

Georg Zitzlsberger
georg.zitzlsberger@vsb.cz

5th of July 2017

Abstract

In this Hands-on session we will optimize a matrix vector multiplication with vectorization. Different exercises show different types of stumbling blocks for vectorization which will be resolved with a broad range of technologies. You'll learn different techniques to increase vectorization quality by this.

1 Getting Started

Follow the steps below for preparation:

1. Login on Salomon and request a compute node (for compilation and execution)

2. The exercises are meant to be executed on Haswell nodes. If you want to use KNC instead, use `-mmic` instead of `-xcore-avx2`; build on the Haswell node and execute on the KNC coprocessor.

3. Load Intel C++ Compiler module:

```
> module load intel/2017.05
> icpc -v
icpc version 17.0.4 (gcc version 7.1.0 compatibility)
```

4. Extract the lab files `Vectorization_Hands_on.tar.bz2` to some folder in your `$HOME`; we'll assume that your working directory is set to this folder.

5. Later, for Activity 3, load:

```
> module load Advisor/2017_update3
> advixe-cl --version
Intel(R) Advisor Command Line Tool
Copyright (C) 2009-2017 Intel Corporation. All rights reserved.
Intel(R) Advisor 2017 Update 3 (build 510716) Command Line Tool
Copyright (C) 2009-2017 Intel Corporation. All rights reserved.
```

It also requires X11 forwarding when starting the GUI via `advixe-gui`!

6. Later, for Activity 4, load:

```
> module load SDE/7.41.0
> sde --version
Intel(R) Software Development Emulator.  Version:  7.41.0 external
Copyright (C) 2008-2015, Intel Corporation. All rights reserved.
```

2 Activity 1 - Vectorization with Intel C++ Compiler

In this activity, you enable the compiler to generate diagnostic information on sample code that does not vectorize initially but can be vectorized, as will be seen in the very end. We start with a first analysis of what is keeping the compiler from vectorization.

2.1 Baseline

The example is contained in the subdirectory `matvec`.
We start with comparing the vectorized against the non-vectorized version. First we compile and run it non-vectorized:

```
> icc -O2 -xcore-avx2 -no-vec -no-simd -qno-openmp-simd multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Now let's see what the vectorization of the compiler can get us:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Question: What is your conclusion?

2.2 Start with Algorithmic Optimizations

Before starting to optimize an application or subcomponents, it should be started at the algorithmic level. Selection of algorithms impacts data flow and computation at a higher level. No compiler can automatically find an alternative algorithm that would perform better - it's subject to research to do this. We exemplify this by the following exercise.
Assume we had extracted the underlying example code from a bigger application. We would find a major limiting factor in terms of memory accesses and unnecessary generalisation. Can you spot it? Can you think of an improvement that works for our case?
Hint: `multiply.c` does not have information from `driver.c` because the compiler translates both independently.

Solution: `solutions/unit-stride`

After the (rather trivial) algorithmic optimizaion has been applied, measure again with vectorization on:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

You might see a slight improvement.
Question: What is your conclusion?

2.3 Compiler Optimization Report

Now, we take a look at what the compilers can tell us about the optimizations. Take a look at vectorization of `multiply.c` not done:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector -qopt-report=5
> cat multiply.optrpt
...
LOOP BEGIN at solutions/unit-stride/multiply.c(36,9)
    remark #15344: loop was not vectorized: vector dependence prevents vectorization
    remark #15346: vector dependence: assumed FLOW dependence between b[i] (37:13) and b[i] (37:13)
    remark #15346: vector dependence: assumed ANTI dependence between b[i] (37:13) and b[i] (37:13)
    remark #25439: unrolled with remainder by 2
LOOP END
...
```

Question: What do those reports mean? What are the root causes?

2.4 Make it Vectorize

In this activity, you will make huge progress with vectorization by remedying the reported problems from the optimization reports. As can be seen, we can solve those in three different ways.

2.4.1 No Aliasing

First, we start by turning off aliasing for all function arguments throughout a whole compilation unit. This is meaningful if we can guarantee that no pointers in all the function arguments overlap (even if they're of the same type, which still is allowed by strict ANSI aliasing). For our example this is the case because all arrays have disjunctive memory locations. Hence, we can compile the code as is without any modifications, only by adding one single compiler option:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector -fargument-noalias
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Compare the optimization outputs with and without this option.

2.4.2 No Vector Dependence

Instead of globally addressing assumed dependencies, we can also break them up on a per-loop basis by using `#pragma ivdep`. Apply it to the proper loop in the code, compare the performance and consult the optimization reports. This time, no additional compiler options are needed:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Solution: `solutions/ivdep`

2.4.3 Restrict Keyword

Finally, we try yet another approach which is even finer grained: We apply the keyword `restrict`. Can you find the correct location where to apply it to? Be aware that we either have to assert C99 conforming code (`-std=c99`) or apply the compiler option `-restrict`. The latter is compiler implementation specific and not standardized. However, most compilers allow that for pre-C99 or even C++ (only for pointers, not references).
When applied correctly, the effective performance should be similar to the previous two solutions:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector -restrict
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Solution: `solutions/restrict`

Consult the optimization reports once more for validation.

2.5 Alignment

Select any of the three solutions from the previous activity above and generate the assembly (`-S`) of `multiply.c`. You can see that the compiler generated multiple versions and tests right in the beginning of the function. Those are caused by unknown alignment of the array elements. Also the optimization reports tell you that.
In the current activity we continue with the solution for `#pragma ivdep` but the others work as well, provided that the required compiler options are

specified in addition. You will improve the performance of the code generated by asserting alignment of data now.

2.5.1 Aligning

The only data that is used throughout the loops are arrays `a`, `b` and `x`. Looking at their allocation in `driver.c`, it turns out that the arrays are not necessarily aligned. By using a simple attribute you can guarantee alignment of all three arrays. Which one would be best here? Change the code accordingly.
The execution time won't change yet:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Hint: `__declspec(align(...))` or `__attribute__((aligned(...)))`
Solution: `solutions/align`

This change was mostly a preparation for the next step, where we are going to use that alignment.

2.5.2 Using Alignment

Next, we look into `multiply.c`. As it is compiled separately (like any compilation unit) it does not have knowledge about the alignment. Thus it has to assume unaligned data accesses. The compiler can generate multiple versions (e.g. for aligned and unaligned access) and select the proper execution path during run-time. This involves some overhead (and increases code size). Since we guaranteed proper alignment for the memory locations, to which all the pointers in the function arguments refer to, we can safely assert this for the compiler. There is a simple way to do so, which one?
The execution time now is reduced:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Hint: `__assume_aligned(...)`
Solution: `solutions/assume_aligned`

2.5.3 Padding for Alignment

Moving our focus away from sole compiler optimizations there is more room for improvement: The amount of elements per row is not multiple of what a

SIMD vector (or multiple thereof) can keep. Hence each row needs remainder handling. This remainder handling causes additional overhead at the end of each row. To avoid this overhead, pad the row to a multiple of the SIMD vector length. Padding is controlled via `COLBUF` in our example. Understand how it works and apply a correct value to it so it is done correctly. Once `COLBUF` is set correctly another improvement should become visible:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 64
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Note: We increased the size of the matrix (2-dimensional array) and yet the performance became much better!

Solution: `solutions/padding`

2.5.4 Enforcing Aligned Accesses

Please ensure that all elements per row are now multiple of the SIMD vector length. This allows a more aggressive optimization regarding alignment using `#pragma vector aligned`. In addition we can tell the compiler that the rows are multiple of at least the vector length by using `__assume(<variable> % <multiple_vl> == 0)`. The value of `<multiple_vl>` can be any multiple of the vector length.
Apply both to correct locations in multiply.c and measure the execution time once more:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 64
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Hint: Only one loop needs the pragma; assertion of elements per row should be as large as possible Solution: `solutions/vector_aligned`

Note:
Using the pragma and enforcing aligned accesses, unconditionally applies to all accesses inside the loop. If we had not padded the matrix before and used the pragma, the compiler would create incorrect code. The reason is that the first row in the array `a` starts at a vector aligned address but the second one does not (elements per row are not multiple of the vector length).
We won't always see a crash for our example because the compiler tends to use unaligned moves which work for both aligned and unaligned data (not for the coprocessor!). Sandy Bridge based processors and later can figure out actual alignment during run-time and, for the case of actually aligned data,

use the much faster aligned accesses internally instead. However, there is no guarantee for that and it has to be expected to face SEGVs when the pragma is used incorrectly. The SEGVs result from the GP faults of instructions that require aligned data but have been provided with unaligned memory references.

Note:
When using the official solution, you will notice that `__assume(cols % 64 == 0)` is used. This is not multiple of 64 byte but a multiple of 64 elements of double precision FP and hence 64 * 8 byte = 512 byte per row. This is also a multiple of the possible vector lengths. The larger the guaranteed elements per vector can be asserted, the more flexibility is granted for the compiler. You could also try to just use the vector length itself. Then you might notice good but still not optimal results.

2.6 Final Comparison

Finally, compare the best result from above against it being not vectorized:

```
> icc -O2 -xcore-avx2 -no-vec -no-simd -qno-openmp-simd multiply.c driver.c -o matvector
> ./matvector
ROW: 64 COL: 64
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Question: What is the overall speedup?

3 Activity 2 - Vectorization with OpenMP 4.0+

In this activity, you use the previous example from Activity 1 and vectorize it by using OpenMP 4.0. Using OpenMP 4.5 extensions is possible but not needed here.

3.1 Enabling OpenMP

We start with using the original version without any improvements and apply OpenMP 4.0 to it. As a pre-requisite, it requires a compiler option `-qopenmp` or `-qopenmp-simd` to be specified. The latter is used here because it does not introduce an OpenMP runtime like the former one and just enables the SIMD constructs.
To the original version with algorithmic optimizations from section 2.2, add both OpenMP SIMD construct and SIMD declare construct. If you've done so, you should see a speedup compared to the original version:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector -qopenmp-simd
> ./matvector
ROW: 64 COL: 63
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Hint: Transform inner loop; don't forget to take the non-unit stride into account (not necessary though)
Solution: `solutions/openmp4_simd`

3.2 Further Optimizations

The vectorization is still not optimal as we've learned in the beginning activity. Now re-apply those improvements to the current version too. Validation of performance should show the same best result as from the previous activity, but using SIMD vectorization of OpenMP 4.0 now:

```
> icc -O2 -xcore-avx2 multiply.c driver.c -o matvector -qopenmp-simd
> ./matvector
ROW: 64 COL: 64
Elapsed time = ??? seconds
GigaFlops = ???
Sum of result = 77172000000.000000
```

Hint: Apply the same steps as in previous activities. Not all might be needed, though. `#pragma ivdep` won't work here because it cannot be combined with SIMD-enabled functions.
Solution: `solutions/openmp4_simd_best`

3.3 SIMD Construct

In this activity, you become familiar with the OpenMP SIMD construct to vectorize code the compiler won't by default and even strict language interpretation won't allow.
This example is rather artificial but demonstrates most of the SIMD construct clauses from OpenMP 4.0 in one combined example.

Note: The examples need to be compiled with compiler option `-qopenmp` or `-qopenmp-simd` set. Otherwise the pragma will be ignored. The latter should be used here because it does not introduce an OpenMP runtime like the former one and just enables the SIMD constructs.

3.3.1 Baseline

The example is contained in the subdirectory `simd`.
Initially we compile our example without any modification and record the execution time and result:

```
> icc -O2 -xcore-avx2 main.c simd.c -o simd
> ./simd
Elapsed time = ??? seconds
Sum: 1249237760.000000
```

Note: Keep an eye on the value of the result shown as *Sum*!

3.3.2 Vectorize

It is quite slow and could be much faster. Hence, this is a great opportunity to take a look at the optimization report:

```
> icc -O2 -xcore-avx2 main.c simd.c -o simd -qopt-report=5
```

Look at the implementation to understand which problems are reported. Note, that it's not important to understand what's computed but more, how it is done.
Can you see were we're having potential for vectorization? Use `#pragma omp simd` to enforce vectorization here.
What do you see and why? What is the execution time and result?

```
> icc -O2 -xcore-avx2 main.c simd.c -o simd -qopenmp-simd
> ./simd
Elapsed time = ??? seconds
Sum: 1249237760.000000
```

Hint: Notice compiler warning
Solution: `solutions/openmp4_simd`

3.3.3 Reduction

What (obviously) needs to be done to safely vectorize the loop in `simd.c` in first place? Which is the critical variable and what's its operation? Apply this change and record the execution time and result:

```
> icc -O2 -xcore-avx2 main.c simd.c -o simd -qopenmp-simd
> ./simd
Elapsed time = ??? seconds
Sum: 22013995008.000000
```

Hint: Take a look at the optimization report
Solution: `solutions/openmp4_reduction`

The result is not correct. There are two other properties of the loop body that need to be provided to the pragma:

- Access via a pointer that's linearly incremented
- Safe max. vector length to be used (hidden in the semantics of the example!)

In the following, two additional clauses are applied to the pragma to retrieve the correct result and take advantage from (enforced) vectorization.

3.3.4 Linear

Identify the pointer which is linearly incremented. By which value?
Apply the corresponding clause to the pragma and compare execution time and result once more:

```
> icc -O2 -xcore-avx2 main.c simd.c -o simd -qopenmp-simd
> ./simd
Elapsed time = ??? seconds
Sum: 22013995008.000000
```

Hint: See comment
Solution: `solutions/openmp4_linear`

3.3.5 Safelength

Finally, we have to assert that the maximum vector length may not exceed a certain value. Otherwise, the result is incorrect which we saw with the previous steps already. This is only true if vector length automatically selected by the compiler together with `#pragma omp simd` exceeds that value! Hence this won't be visible with SSE but is visible with AVX and Intel MIC architecture (single precision FP is used!).
Which vector length is safe and how can it be asserted?
Apply the corresponding clause to the pragma and compare execution time and result a last time:

```
> icc -O2 -xcore-avx2 main.c simd.c -o simd -qopenmp-simd
> ./simd
Elapsed time = ??? seconds
Sum: 1249237760.000000
```

The result is correct now. What is the speedup compared to the initial (compiler only) version?

Hint: See comment
Solution: `solutions/openmp4_safelength`

4 Activity 3 - Quality of Vectorization

In this activity, we revisit the previous examples from activities 1 & 2 and analyze the vectorization efficiency and memory access patterns with *Intel Advisor*. We also visualize the hot loops in the roofline model.
Start the tool with `advixe-gui`, create a project and analyze the previous examples.

4.1 Vectorization Efficiency

Measure the vectorization efficiency of the baseline implementations and the final ones. What is the change of the efficiency?

4.2 Roofline Model

Show the roofline model for the baseline implementations and the final ones. How do they change?

4.3 Memory Access Patterns

Even in the baseline implementations, the memory accesses are unit-strided. However, change the order of the inner and outer loops in `multiply.c` and observe the change in the memory access patterns output.

5 Activity 4 - Future of Vectorization

In this activity, we use *Intel Software Development Emulator* (SDE) to emulate AVX-512 enabled applications on Haswell. We again use the examples from activities 1 & 2.

5.1 Build with AVX-512

Build the baseline of both examples from activities 1 & 2 as well as their best implementations with AVX-512. It's up to you whether you use KNL (`-xmic-avx512`) or Skylake AVX-512 (`-xcore-avx512`).

5.2 Run Applications with SDE

Run the applications with SDE for KNL, e.g.:

```
> sde -knl -- ./matvec
```

...or for Skylake (Server), e.g.:

```
> sde -skx -- ./matvec
```

Note: Don't use `-skl` because that's Skylake client which does not have AVX-512 extensions!

Question: Use the histogram and mask profiling tools of SDE to get information about how many AVX-512 instructions were executed and whether masking was used.