Vectorization: Writing C/C++ code in VECTOR Format

Mukkaysh Srivastav
Computational Research Laboratories (CRL) – Pune, India

1.0 Introduction: Vectorization has been key optimization principle over x87 stack more than a decade. But often C/C++ algorithmic source-code is written without adequate attention to vectorization concepts. Since the performance of many high performance computing (HPC) scientific-applications degrades on multi-core architecture (MCA) (E.g.: Intel® Xeon® 5355 processor) during execution, writing most compute part of algorithm in *vector-length* format would enhance the performance of such scientific-applications by vectorizing the source code either in 2 Double-precision (8-byte) or 4 Single-precision (4-byte) or 16-int (1-byte) *vector-length* format within 128-bit SSE stacks. Optimizing programs for modern multiprocessor or vector platforms is a major important challenge for Compilers today. Writing C/C++ code in a form that will enable a good C/C++ compiler to vectorize it is likely to yield some results in performance improvement. In this article, focus has been made in re-designing C/C++ code in *vector-length* format to address SSE needs. The source code used to demonstrate re-designing of compute algorithm is from multi-file C/C++ scientific-applications HPC package namely, AutoDock (http://autodock.scripps.edu/).

1.1 Vectorization Overview: Vectorization is a special case of SIMD, a term defined in Flynn's architecture taxonomy to denote a single instruction stream capable of operating on multiple data elements in parallel. The number of elements which can be operated on in parallel range from four single-precision floating point data elements in Streaming SIMD Extensions and two double-precision floating-point data elements in Streaming SIMD Extensions 2 to sixteen byte operations in a 128-bit register in Streaming SIMD Extensions 2. Thus, *vector-length* ranges from 2 to 16, depending on the instruction extensions used and on the data type.

Vectorization has been currently used both as auto-vectorization and explicit vectorization, namely Single Instructions Multiple Data (SIMD) or simdization. Producing SIMD codes is sometimes done manually for important specific application, but is often produced automatically by Compilers (referred to as auto-vectorization). Auto-vectorization is enabled through compiler flags. Compiler flags for auto-vectorization can be referred from the relevant manuals, like *Intel C++ Compiler User & Reference Guides*. An auto-vectorizing compiler parallelizes code to utilize streaming SIMD extensions (SSE) instruction set architectures (SSE, SSE2, SSE3, SSSE3, and SSE4) of latest processors. Explicit form of vectorization, SIMD is used either as Inline assembly, intrinsic or pure assembly .S representation. Explicit vector programming is time consuming, error prone and also machine dependent, so objective should be in re-designing the code in *vector-length* format. Moreover, simdization is not trivial, it has been observed that difficulties in optimizing code for SIMD architecture stems from hardware constraints too. Intel C++ Compiler supports GNU*-Assembler (GAS) syntax on Linux* x86_64 as default assembler, so writing either GNU-syntax Inline assembly or pure .S assembly file addressing SSE would be a tough job rather re-designing C/C++ compute code.

1.2 C/C++ Code Format: The source code as demonstrated here has been taken from one of the C++ files of *AutoDock (v-4.0.1)* scientific-applications package as available under GPL. *AutoDock* is a suite of automated docking tools; it has been designed to predict how small molecules, such as substrates or drug candidates, bind to a receptor of known 3D structure. One of the *AutoDock (v-4.0.1)* C++ source code file *torsion.cc* file has been vectorized and described here. This *torsion.cc* file API was chosen because this API has high CPU processing time as generated through use of Intel VTune profiler. More details about AutoDock can be obtained from http://autodock.scripps.edu/. The original compute algorithm is -

```
crd[mvatm][X] = (double)crdtemp[X] + d[X] * k[X][X] + d[Y] * k[X][Y] + d[Z] * k[X][Z];
crd[mvatm][Y] = (double)crdtemp[Y] + d[X] * k[Y][X] + d[Y] * k[Y][Y] + d[Z] * k[Y][Z];
crd[mvatm][Z] = (double)crdtemp[Z] + d[X] * k[Z][X] + d[Y] * k[Z][Y] + d[Z] * k[Z][Z];
```

The above code can't be vectorized because the loop contains three parameters (X, Y & Z is defined in *autocomm.h* file of AutoDock *v-4.0.1* package as macros for X, Y & Z co-ordinates) but double-precision (DP) floating-point (FP) SIMD elements wants data either in 2, or 4 *vector-length* format, and moreover the algorithm access pattern through k is *non-unit stride*. The original code if executed with Intel C++ Compiler (ICC-v11.0) with *-vec-report3* flag generates lots of data dependencies hazards as -

```
torsion.cc(106): (col. 15) remark: vector dependence: assumed ANTI dependence between crd line 106 and crd line 109.
torsion.cc(109): (col. 15) remark: vector dependence: assumed FLOW dependence between crd line 109 and crd line 106.
torsion.cc(106): (col. 15) remark: vector dependence: assumed ANTI dependence between crd line 106 and crd line 109.
torsion.cc(109): (col. 15) remark: vector dependence: assumed FLOW dependence between crd line 109 and crd line 106.
torsion.cc(106): (col. 15) remark: vector dependence: assumed ANTI dependence between crd line 106 and crd line 109.
                                        .....
                                        .....
torsion.cc(108): (col. 15) remark: vector dependence: assumed FLOW dependence between crd line 108 and crd line 105.
torsion.cc(104): (col. 15) remark: vector dependence: assumed ANTI dependence between crd line 104 and crd line 107.
torsion.cc(107): (col. 15) remark: vector dependence: assumed FLOW dependence between crd line 107 and crd line 104.
torsion.cc(104): (col. 15) remark: vector dependence: assumed ANTI dependence between crd line 104 and crd line 107.
torsion.cc(107): (col. 15) remark: vector dependence: assumed FLOW dependence between crd line 107 and crd line 104.
                                        ....
                                        ....
torsion.cc(109): (col. 15) remark: vector dependence: assumed FLOW dependence between crd line 109 and tlist line 103.
torsion.cc(103): (col. 15) remark: vector dependence: assumed ANTI dependence between tlist line 103 and crd line 109.
torsion.cc(109): (col. 15) remark: vector dependence: assumed FLOW dependence between crd line 109 and tlist line 103.
torsion.cc(103): (col. 15) remark: vector dependence: assumed ANTI dependence between tlist line 103 and crd line 109.
torsion.cc(109): (col. 15) remark: vector dependence: assumed FLOW dependence between crd line 109 and tlist line 103.
```

Since from above generated compiler compiled message we see that code has too many *flow* & *false dependencies*. Our objective is to demonstrate in getting rid of both *flow* & *false* dependencies and have unit-stride access pattern.

These problem of above code not been vectorized, aligned and having unit-stride access pattern has been demonstrated in *section 1.4* by simply re-designing *torsion.cc* file of AutoDock by addressing *vector-length* format.

1.3 SIMD Non-vectorized Assembly Behavior: The behavior of SSE assembly instructions for above algorithm has been –

```
movaps      %xmm5, %xmm12
mulsd       %xmm15, %xmm12
addsd       %xmm2, %xmm12
movaps      %xmm9, %xmm0
mulsd       %xmm14, %xmm0
addsd       %xmm0, %xmm12
movaps      %xmm11, %xmm0
mulsd       %xmm13, %xmm0
```

```
addsd       %xmm0, %xmm12
cvtsd2ss    %xmm12, %xmm12
movss       %xmm12, (%r10,%rdi)

movsd       40(%rsp), %xmm0
mulsd       %xmm15, %xmm0
addsd       %xmm4, %xmm0
movaps      %xmm6, %xmm12
mulsd       %xmm14, %xmm12
addsd       %xmm12, %xmm0
movaps      %xmm7, %xmm12
mulsd       %xmm13, %xmm12
addsd       %xmm12, %xmm0
cvtsd2ss    %xmm0, %xmm0
movss       %xmm0, 4(%r10,%rdi)

mulsd       %xmm8, %xmm15
addsd       %xmm3, %xmm15
mulsd       %xmm10, %xmm14
addsd       %xmm14, %xmm15
mulsd       %xmm1, %xmm13
addsd       %xmm13, %xmm15
cvtsd2ss    %xmm15, %xmm13
movss       %xmm13, 8(%r10,%rdi)
```

From the above SSE assembly code, we see alignment didn't happen for *crd[mvatm][Y]* and *crd[mvatm][Z]* expressions. Also, it is seen that **movsd** for *k[Y][X]* is scalar code, so the alignment is a lot less important (since floats are usually 4-byte aligned; for vector data we want the more strict 16-byte alignment). Data dependencies could have been resolved if data rearrangement **unpack (unpcklpd, etc.)** instructions could have been invoked by the compiler along with **mulpd**, **addpd**, **movhpd** instructions. Moreover, due to associated overhead of data rearrangement, it is generally profitable to vectorize only those loops that have relatively few non-unit-stride memory references.

Understanding of above assembly (.S) content can be done either by using ICC (*v-11.0.074*) to generate above assembly code (.S) for *torsion.cc*, which does generate .S file with line number in aligned with source-code line number or by having *objdump* for AutoDock (*v-4.0.1*) executable. Interpreting further the SSE instructions can be done by referring to *Intel 64 and IA-32 Architectures Optimization Reference Manual* (*Order Number: 248966-018*) document.

1.4 C/C++ Vectorized Code Format: Having the problem as discussed in *section 1.2 & 1.3*, we re-design the *torsion.cc* API code such that *torsion.cc* compute algorithm becomes amicable to unit-stride and vectorization. In original code we have X, Y & Z as three parameters, function arguments being passed with array layout of *crd [2048][3]*, and local variable array layout for k being [3][3]. Since, SSE are 128-bit width registers which means it should either have 4 single-precision (SP) floating-point (FP) or 2 double-precision (DP) floating-point (FP) to fit equally in 128-bit both for alignment and effective vectorization. Since this mis-alignment prohibits vectorization, we introduce one dummy “S” parameter which provides the correct alignment for the 128-bit SSE register and enables vectorization. To have unit-stride access of memory, we transpose original k matrix. Also, we increase the size of k matrix to ensure proper mapping of 4th. parameter. Since an array of layout of *crd [2048][3]* is passed as function argument, so to copy this *crd* array layout with local increased array layout of *crdmod[2048][4]*, we use *memcpy()* operation than any other programming techniques of array copy operation as *memcpy()* is most efficient way to perform copy of arrays, we prefer it to use here than any other techniques. Having the array layout now in SSE-aligned format, the modified compute algorithm of *torsion.cc* is –

```
/* Incorporate S as fourth parameter along with transposed k in 2 DP FP format */
/* Below statements are vectorized */
crdmod[mvatm][X] = (double)crdtemp[X] + d[X] * k[X][X] + d[Y] * k[Y][X] + d[Z] * k[Z][X];
crdmod[mvatm][Y] = (double)crdtemp[Y] + d[X] * k[X][Y] + d[Y] * k[Y][Y] + d[Z] * k[Z][Y];
crdmod[mvatm][Z] = (double)crdtemp[Z] + d[X] * k[X][Z] + d[Y] * k[Y][Z] + d[Z] * k[Z][Z];
crdmod[mvatm][S] = (double)crdtemp[S] + d[X] * k[X][S] + d[Y] * k[Y][S] + d[Z] * k[Z][S];
```

Compiling above algorithm as -

```
[@n1 autodock-4.0.1]$ icpc -g -O0 torsion.cc -S

[@n1 autodock-4.0.1]$ icpc -g -O1 torsion.cc -S

[@n1 autodock-4.0.1]$ icpc -g -O2 torsion.cc -S
 torsion.cc(128): (col. 15) remark: BLOCK WAS VECTORIZED.

[@n1 autodock-4.0.1]$ icpc -g -O3 torsion.cc -S
 torsion.cc(128): (col. 15) remark: BLOCK WAS VECTORIZED.

[@n1 autodock-4.0.1]$ icpc torsion.cc -S
torsion.cc(128): (col. 15) remark: BLOCK WAS VECTORIZED.
```

We observe that, Intel C++ Compiler (*ICC v-11.0.074*) vectorizes both with -O2, -O3 and ICC default optimization (-O2) but not with -O1 & -O0 for *torsion.cc* modified code of AutoDock which was desired finally and moreover we don't see any compiler generated dependency messages as seen in *section 1.2* for un-modified compute algorithm having X, Y & Z parameters. Current ICC-v11.0 supports vectorization at -O2 & aggressive vectorization at -O3 optimization levels.

Here, we have seen that ICC (*v-11.0.074*) successfully generates vectorization message as "BLOCK WAS VECTORIZED" but not as "LOOP WAS VECTORIZED". Sometime we get confused with ICC vectorization messages between "LOOP WAS VECTORIZED" or "BLOCK WAS VECTORIZED. The only difference between message of types "LOOP WAS VECTORIZED" or "BLOCK WAS VECTORIZED" can be understood with below example -

```
for (int i = 0; i < 2; i++)
     a[i] = b[i] + 1;
```

gives LOOP WAS VECTORIZED at the line number of the FOR loop, while the equivalent

```
a[0] = b[0] + 1;
a[1] = b[1] + 1;
```

gives BLOCK WAS VECTORIZED at the line number of the first statement. This difference of messages shouldn't generate confusion in interpreting vectorization messages by ICC.

1.5 SIMD Vectorized Assembly Behavior:

Below assembly is representation of being vectorized for modified algorithm with 4th. parameter "S" -

```
movaps          %xmm14, %xmm0
movhpd          24(%rsp), %xmm0
movaps          %xmm6, %xmm4
```

```
                unpcklpd      %xmm9, %xmm4

                movhpd        56(%rsp), %xmm13
                unpcklpd      %xmm0, %xmm5
                movaps        %xmm3, %xmm14
                movhpd        88(%rsp), %xmm12
                unpcklpd      %xmm7, %xmm3

                unpcklpd      %xmm8, %xmm2
                movhpd        120(%rsp), %xmm1

...B1.6:                                                          // Level
                movaps        %xmm5, %xmm9
                mulpd         %xmm8, %xmm9
                mulpd         %xmm13, %xmm8
                addpd         %xmm4, %xmm9
                unpcklpd      %xmm15, %xmm15
                addpd         %xmm0, %xmm8

                movaps        %xmm3, %xmm14
                mulpd         %xmm15, %xmm14
                mulpd         %xmm12, %xmm15
                addpd         %xmm14, %xmm9
                unpcklpd      %xmm7, %xmm7
                addpd         %xmm15, %xmm8

                movaps        %xmm2, %xmm14
                mulpd         %xmm7, %xmm14
                mulpd         %xmm1, %xmm7
                addpd         %xmm14, %xmm9
                cvtpd2ps      %xmm9, %xmm9
                addpd         %xmm7, %xmm8
                cvtpd2ps      %xmm8, %xmm7
                movlhps       %xmm7, %xmm9
                movaps        %xmm9, 160(%rsp,%rdi)
                unpcklpd      %xmm8, %xmm8
```

We see from above that **unpcklpd** instruction has been called which does resolves dependencies. Also, when using **movlhps** the data it takes from immediate "**cvtpd2ps %xmm8, %xmm7**" operations is from low quad-word of **xmm7** thus **movlhps** uses only the lower part of **xmm7**; these two instructions are functionally independent. However **cvtpd2ps** will "lock" **%xmm7** till it completion thus preventing **movlhps**. Moreover, **addpd** & **mulpd** has been used here than earlier non-vectorized assembly code of *section 1.3* where we saw **addsd** and **mulsd** instructions calls. We also see that **movhpd** instructions have been used to perform the load/store since compiler cannot provide alignment of data. The use of **movhpd** instruction has a significant impact on the performance finally.

As, Intel Xeon processor has 128-bit-wide floating point unit (4 flops/cycle), it becomes more important to use full-width SSE2 instructions as opposed to the "upper/lower" half varieties if data alignment is known. In other words, instructions such as **mulpd**, **addpd,** etc. are preferable than instructions **mulsd**, **addsd**, etc., as obtained in *section 1.3* because aligned data and vectorizable code are more beneficial and moreover these instructions has low latency than lower quad-word double-precision (DP) floating-point (FP) operation.

1.6 Conclusion:

We therefore conclude that if we know how to write any C/C++ source-code compute algorithm in *vector-length* format to address current SSE stacks of multi-core architectures, compiler for this algorithm can automatically vectorizes the code or in other way compiler job of performing vectorization becomes much easier. We also see that by having *vector-length* formatted C/C++ code, compiler generates and uses more efficient vectorizable instructions (like **unpcklpd**, **addpd**, **mulpd**, **movlhps**, **movhpd**, etc.) which do take care of dependencies.

The above modified C/C++ compute algorithm source-code for *torsion.cc* of AutoDock (*v-4.0.1*) package has been demonstrated only on Intel® Xeon® 5355 processor. This approach of writing C/C++ source-code in *vector-length* format can be true in any other processors which support SIMDization.

Somehow, not verified if Intel C++ Compiler (*ICC v-11.0.074*) can generate **SHUFPD** instructions which can yield better results in place of **unpcklpd**. Also, ICC generated assembly for modified code haven't been tried further either as optimized Inline assembly API nor as pure optimized .S assembly file and latter to use **SHUFPD** in place of **unpcklpd** instructions for having much better performance.

Above exercise of having *vector-length* format of C/C++ source-code can easily be executed on Linux x86_64 machine using *torsion.cc* C++ file of *AutoDock* (*v-4.0.1*) source-code as it is easily available under GPL.

1.7 References:

1. The Software Vectorization Handbook: By Aart J.C. Bik, Intel Press
2. Optimizing Compilers for Modern Architectures: By Randy Allen and Ken Kennedy
3. Intel 64 and IA-32 Architectures Optimization Reference Manual (Order Number: 248966-018)
4. AutoDock Source code: http://autodock.scripps.edu/

1.8 System Configuration:

- Intel C++ Compiler:

  Intel(R) C++ Intel(R) 64 Compiler Professional for applications running on Intel(R) 64, Version 11.0 Build 20081105 Package ID: l_cproc_p_11.0.074

- Operating System:

  Linux n1 2.6.9-55.9hp.4sp.XCsmp #1 SMP Tue Nov 27 18:32:01 EST 2007 x86_64 x86_64 x86_64 GNU/Linux

- CPU Information:

```
vendor_id        : GenuineIntel
model name       : Intel(R) Xeon(R) CPU        X5355  @ 2.66GHz
cache size       : 4096 KB
cpu cores        : 4
fpu              : yes
fpu_exception    : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
                   acpi mmx fxsr sse sse2 ss ht tm syscall lm pni monitor ds_cpl est tm2 cx16 xtpr
bogomips         : 5337.26
clflush size     : 64
cache_alignment : 64
address sizes    : 36 bits physical, 48 bits virtual
```

- Memory Information:

```
Memory Total     : 16413224 kB
Page Tables      : 1428 kB
Vmalloc Total    : 536870911 kB
CPU              : L1 I cache: 32K, L1 D cache: 32K
CPU              : L2 cache: 4096K
```

- Linux Development Tool Components:

```
GNU C  (gcc --version)
gcc (GCC) 3.4.6 20060404 (Red Hat 3.4.6-8)

GNU C++ (g++ --version)
g++ (GCC) 3.4.6 20060404 (Red Hat 3.4.6-8)

Binutils (as -v)
GNU assembler version 2.15.92.0.2 (x86_64-redhat-linux) using BFD version 2.15.92.0.2 20040927

GLIBC  (/lib/libc-2.3.4.so)
GNU C Library stable release version 2.3.4, by Roland McGrath et al.

LIBSTDC++ (/usr/lib/libstdc++.so.6.0.3)
libstdc++.so.6.0.3
```

1.9 Glossary:

HPC	High Performance Computing
Vector-Length	The number of data elements packed together
Precision	The number of bits per data element
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extension Intel instruction set multimedia extension. Versions: SSE (1999), SSE2 (2001), SSE3 (2004), SSSE3 (2005), SSE4 (2006).
ICC	Intel C++ Compiler
GAS	GNU Assembler

2.0 Acknowledgment: I am thankful to **Aart J.C. Bik** of Google (USA) for bringing excellent book on Vectorization (The Software Vectorization Handbook), **Dr. Parag Prasad** (Life-Science Group, Computational Research Laboratories - India) in allowing me to investigate and optimize AutoDock source code, **Milind Athavale** (System Software Group, Computational Research Laboratories - India) and **Shreeniwas Sapre** (Library Group, Computational Research Laboratories - India) for reviewing this document.

2.1 About the author: Author, Mukkaysh Srivastav works with Computational Research Laboratories (CRL) – Pune, India. His areas of interest are Compiler development, HPC Software Tools, Static-Analysis (mainly MPI), Performance & Optimization, SIMD x86_64 Assembly Programming and Public Speaking. With academic qualification, he has been affiliated with University of Massachusetts – Boston (USA); Indian Institute of Technology – Delhi (India) and Regional Engineering College – India. He can be reached through srimks@msn.com.

*****