Processor Specific Optimization

Part 3: Programming for SIMD

J. Eitzinger

PATC LRZ, 18.3.2019
History of short vector SIMD


64 bit 128 bit 256 bit 512 bit

Vendors package other ISA features under the SIMD flag:
monitor/mwait, NT stores, prefetch ISA, application specific instructions, FMA

X86
Intel 512bit
AMD 256bit

Power
IBM 128bit

ARM
A5 upward
64bit / 128bit

SPARC64
Fujitsu 256bit
K Computer
128bit
SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n;i++)
    C[i]=A[i]+B[i];
```

“Loop unrolling”

```
for(int i=0; i<n;i+=4){
    C[i]  =A[i]  +B[i];
    C[i+1]=A[i+1]+B[i+1];
```

//remainder loop handling

```
LABEL1:
VLOAD R0  0x0 A[i]
VLOAD R1  0x0 B[i]
V64ADD[R0,R1]  R2
VSTORE R2  C[i]
ii+4
i<(n-4)? JMP LABEL1
```

//remainder loop handling

Load 256 Bits starting from address of A[i] to register R0
Add the corresponding 64 Bit entries in R0 and R1 and store the 4 results to R2
Store R2 (256 Bit) to address starting at C[i]
SIMD processing – Basics

No SIMD vectorization for loops with data dependencies:

for(int i=0; i<n;i++)
    A[i]=A[i-1]*s;

“Pointer aliasing” may prevent SIMDfication

void f(double *A, double *B, double *C, int n) {
    for(int i=0; i<n; ++i)
        C[i] = A[i] + B[i];
}

C/C++ allows that A \rightarrow \&C[-1] \text{ and } B \rightarrow \&C[-2] 
\rightarrow C[i] = C[i-1] + C[i-2]: \text{ dependency} \rightarrow \text{No SIMD}

If “pointer aliasing” is not used, tell it to the compiler:

- \text{\texttt{-fno-alias}} (Intel), \text{\texttt{-Msafeptr}} (PGI), \text{\texttt{-fargument-noalias}} (gcc)
- \text{\texttt{restrict}} keyword (C only!):

void f(double restrict *A, double restrict *B, double restrict *C, int n) {...}
How to leverage SIMD: your options

Alternatives:
- The **compiler** does it for you (but: aliasing, alignment, language)
- Compiler directives (**pragmas**)
- Alternative **programming models** for compute kernels (OpenCL, ispc)
- **Intrinsics** (restricted to C/C++)
- Implement directly in **assembler**

To use **intrinsics** the following headers are available:
- `xmmmintrin.h` (**SSE**)  
  ```c
  for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
  }
  ```
- `pmmintrin.h` (**SSE2**)  
- `immintrin.h` (**AVX**)  
- `x86intrin.h` (**all extensions**)
Vectorization compiler options (Intel)

- The compiler will vectorize starting with \(-O2\).
- To enable specific SIMD extensions use the \(-x\) option:
  - \(-xSSE2\) vectorize for SSE2 capable machines
  - \(-xAVX\) on Sandy/Ivy Bridge processors
  - \(-xCORE-AVX2\) on Haswell/Broadwell
  - \(-xCORE-AVX512\) on Skylake (certain models)
  - \(-xMIC-AVX512\) on Xeon Phi Knights Landing

Available SIMD extensions:
SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, ... 

Recommended option:
- \(-xHost\) will optimize for the architecture you compile on
  (Caveat: do not use on standalone KNL, use MIC-AVX512)
- To really enable 64b SIMD with current Intel compilers you need to set:
  \(-qopt-zmm-usage=high\)
User mandated vectorization (OpenMP4)

- Since Intel Compiler 12.0 the `simd` pragma is available (now deprecated)
- `#pragma simd` enforces vectorization where the other pragmas fail
- Prerequisites:
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- There are additional clauses: `reduction`, `vectorlength`, `private`
- Refer to the compiler manual for further details

```c
#pragma simd reduction(+:x)
for (int i=0; i<n; i++) {
    x = x + A[i];
}
```

- NOTE: Using the `#pragma simd` the compiler may generate incorrect code if the loop violates the vectorization rules! It may also generate inefficient code!
Why and how?

Why check the assembly code?

- Sometimes the only way to make sure the compiler “did the right thing”
  - Example: “LOOP WAS VECTORIZED” message is printed, but Loads & Stores may still be scalar!

- Get the assembler code (Intel compiler):
  \[ \text{icc} -S -O3 -xHost triad.c -o a.out \]

- Disassemble Executable:
  \[ \text{objdump} -d ./a.out | less \]
Basics of the x86-64 ISA

- Instructions have 0 to 3 operands (4 with AVX-512)
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two assembler syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: \texttt{A[i]} equivalent to \texttt{*(A+i)} (a pointer has a type: \texttt{A+i*8})

\begin{verbatim}
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
\end{verbatim}

\begin{verbatim}
movaps %xmm4, 48(%rdi,%rax,8)
addq $8, %rax
js ..B1.4
\end{verbatim}

\begin{verbatim}
401b9f: 0f 29 5c c7 30 movaps %xmm3,0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08 add $0x8,%rax
401ba8: 78 a6 js 401b50 <triad_asm+0x4b>
\end{verbatim}
16 general Purpose Registers (64bit):
\[ \text{rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15} \]
alias with eight 32 bit register set:
\[ \text{eax, ebx, ecx, edx, esi, edi, esp, ebp} \]
8 opmask registers (16bit or 64bit, AVX512 only):
\[ \text{k0–k7} \]

Floating Point **SIMD** Registers:
\[ \text{xmm0-xmm15 (xmm31) } \]
SSE (128bit) alias with 256-bit and 512-bit registers
\[ \text{ymm0-ymm15 (xmm31) } \]
AVX (256bit) alias with 512-bit registers
\[ \text{zmm0-zmm31 } \]
AVX-512 (512bit)

SIMD instructions are distinguished by:
VEX/EVEX prefix: \[ v \]
Operation: \[ \text{mul, add, mov} \]
Modifier: nontemporal (\text{nt}), unaligned (\text{u}), aligned (\text{a}), high (\text{h})
Width: scalar (\text{s}), packed (\text{p})
Data type: single (\text{s}), double (\text{d})
ISA support on Intel chips

SKylake supports all **legacy** ISA extensions:

**MMX, SSE, AVX, AVX2**

Furthermore **KNL** supports:

- AVX-512 Foundation (F), KNL and Skylake
- AVX-512 Conflict Detection Instructions (CD), KNL and Skylake
- AVX-512 Exponential and Reciprocal Instructions (ER), KNL
- AVX-512 Prefetch Instructions (PF), KNL

AVX-512 extensions only supported on **Skylake**:

- AVX-512 Byte and Word Instructions (BW)
- AVX-512 Doubleword and Quadword Instructions (DQ)
- AVX-512 Vector Length Extensions (VL)

**ISA Documentation:**

*Intel Architecture Instruction Set Extensions Programming Reference*
ISA support on KNL

KNL supports all legacy ISA extensions:
**MMX, SSE, AVX, AVX2**

Furthermore **KNL** supports:
- AVX-512 Foundation (F), KNL and Skylake
- AVX-512 Conflict Detection Instructions (CD), KNL and Skylake
- AVX-512 Exponential and Reciprocal Instructions (ER), KNL
- AVX-512 Prefetch Instructions (PF), KNL

AVX-512 extensions only supported on **Skylake**:
- AVX-512 Byte and Word Instructions (BW)
- AVX-512 Doubleword and Quadword Instructions (DQ)
- AVX-512 Vector Length Extensions (VL)

**ISA Documentation:**
*Intel Architecture Instruction Set Extensions Programming Reference*
Limits of SIMD processing

- Only part of application may be vectorized, arithmetic vs. load/store (Amdahls law), data transfers
- Memory saturation often makes SIMD obsolete

<table>
<thead>
<tr>
<th></th>
<th>Execution</th>
<th>Cache</th>
<th>Memory</th>
<th>AVX example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>16cy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSE</td>
<td>4cy</td>
<td>4cy</td>
<td>4cy</td>
<td></td>
</tr>
<tr>
<td>AVX</td>
<td>2cy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVX512</td>
<td>1cy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Possible solution: Improve cache bandwidth

Total runtime with data loaded from memory:
- Scalar 24
- SSE 12
- AVX 10
- AVX512 9

Per-cacheline (8 iterations) cycle counts
Example for masked execution

Masking for predication is very helpful in cases such as e.g. remainder loop handling or conditional handling.
Case Study: Simplest code for the summation of the elements of a vector (single precision)

```c
float sum = 0.0;

for (int i=0; i<size; i++){
    sum += data[i];
}
```

Instruction code:

```
addss 0(%rdx,%rax,4),%xmm0
add rax,1
cmp edi,eax
ja 401d08
```

To get object code use `objdump -d` on object file or executable or compile with `-S`

AT&T syntax:

```
addss xmm0,[rdx + rax * 4]
add rax,1
cmp edi,eax
ja 401d08
```

(final sum across xmm0 omitted)
Regulations of software interaction on binary level

- Registers \texttt{rbp}, \texttt{rbx} and \texttt{r12} - \texttt{r15} “belong” to the calling function and the called function is required to preserve their values.
- Remaining registers “belong” to the called function.

- The value (\texttt{rsp} + 8) is always a multiple of 16 (32) when control is transferred to the function entry point.

- Pointer or integer arguments are passed the next available register of the sequence \texttt{rdi}, \texttt{rsi}, \texttt{rdx}, \texttt{rcx}, \texttt{r8} and \texttt{r9}.
### Calling subroutines and the stack

```
triad_avx :
push rbp
mov rbp, rsp
push rbx
push r12
push r13
push r14
push r15

Move stack ptr rsp for allocation / deallocation

pop r15
pop r14
pop r13
pop r12
pop rbx
mov rsp, rbp
pop rbp
ret
```

<table>
<thead>
<tr>
<th>Position</th>
<th>Contents</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>8n+16(%rbp)</td>
<td>memory argument eightbyte n</td>
<td>Previous</td>
</tr>
<tr>
<td>16(%rbp)</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8(%rbp)</td>
<td>return address</td>
<td></td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>previous %rbp value</td>
<td></td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td>unspecified</td>
<td></td>
</tr>
<tr>
<td>0(%rsp)</td>
<td>variable size</td>
<td></td>
</tr>
<tr>
<td>-128(%rsp)</td>
<td>red zone</td>
<td></td>
</tr>
</tbody>
</table>

### The operating system provides:

- **Stack:**
  - First in / Last out data structure
  - Provided per thread
  - Grows towards lower addresses

- **Heap:**
  - Free allocation/ deallocation
  - Provided per process
Rules for vectorizable loops

1. Inner loop
2. Countable (loop length can be determined at loop entry)
3. Single entry and single exit
4. Straight line code (no conditionals)
5. No (unresolvable) read-after-write data dependencies
6. No function calls (exception intrinsic math functions)

Better performance with:
1. Simple inner loops with unit stride (contiguous data access)
2. Minimize indirect addressing
3. Align data structures to SIMD width boundary
4. In C use the restrict keyword and/or const qualifiers and/or compiler options to rule out array/pointer aliasing
Documentation

- Intel Intrinsics guide:

- Intel Software Developer Manuals and Software Optimization Reference Manual:

- Linux X86-64 ABI:

Useful tools:
- https://www.gnu.org/software/gdb/
- https://www.gnu.org/software/binutils/