Vectorization

Devin A. Matthews
UT Austin

MolSSI Software Summer School 2017

Vectorization

- In order to perform several floating-point operations at the same time, data must be loaded into **vector registers**.
- The transformation of a program to a form where this is possible is called **vectorization**. This comes in two main forms:
  - Compiler auto-vectorization
    - Completely automatic
    - With help from the programmer
  - Manual vectorization
    - Intrinsics
    - Inline assembly

Loops vs. blocks

- Most compiler auto-vectorization is targeted towards vectorizing **loops**. This generally means that each element in the vector will be a different loop iterations.
- Some compilers (e.g. icpc) can also vectorize **blocks**. This is a piece of code where multiple operations in the same loop iteration can be vectorized.
- You can't rely 100% on any type of compiler auto-vectorization. Always check the **disassembly**!

Enabling vectorization: compiler flags

- In order to vectorize your code, you have to give the compiler the green light through certain flags:

Effect	Flag	Compiler(s)
Enable vectorization	-O3	g++
	-O2	icpc, clang++
Enable a particular instruction set	-msse<xy>, -mavx, -mavx2, -mfma	all
Enable all instruction sets for a particular architecture	-march=haswell, etc.	g++, clang++
	-xCORE-AVX2, etc.	icpc
Enable vectorization for *this* processor.	-march=native	g++, clang++
	-xHost	icpc

Sample 1: DSCAL

Compile with: g++ -O3 -march=native –fno-unroll-loops –c -o sample1.o sample1.cxx

```
void dscal(int n, double alpha, double* x /* assume incx = 1 */)
{
    for (int i = 0;i < n;i++)
    {
        x[i] *= alpha;
    }
}
```

Getting a disassembly

- A **disassembly** is the sequence of machine instructions that the compiler generated for our code. We can get this from an executable or **object file** with:

  - OSX/macOS:

  `otool -vt` *`<executable_or_object_file>`* `[ | less ] [ >` *`<file>`* `]`

  - Linux, WSL:

  `objdump -d` *`<executable_or_object_file>`* `[ | less ] [ >` *`<file>`* `]`

Sample 1 disassembly (no vectorization)

```
00    pushq   %rbp
01    movq    %rsp, %rbp
04    testl   %edi, %edi
06    jle     0x20
08    nopl    (%rax,%rax)
10    vmulsd  (%rsi), %xmm0, %xmm1
14    vmovsd  %xmm1, (%rsi)
18    addq    $0x8, %rsi
1c    decl    %edi
1e    jne     0x10
20    popq    %rbp
21    retq
```

If n==0, quit early

Loop: %rsi == x+i.

Note that this uses SD (single double precison) operations → no vectorization.

Sample 1 disassembly (with vectorization)

```
2f    vmovddup     %xmm0, %xmm1    ## xmm1 = xmm0[0,0]
33    vinsertf128  $0x1, %xmm1, %ymm1, %ymm1
39    incq    %rcx
3c    andq    %r9, %rcx
3f    movq    %rsi, %rdx
42    nopw    %cs:(%rax,%rax)
50    vmovupd (%rdx), %xmm2
54    vinsertf128  $0x1, 0x10(%rdx), %ymm2, %ymm2
5b    vmulpd  %ymm1, %ymm2, %ymm2
5f    vextractf128  $0x1, %ymm2, 0x10(%rdx)
66    vmovupd %xmm2, (%rdx)
6a    addq    $0x20, %rdx
6e    addq    $-0x4, %rcx
72    jne     0x50
```

Broadcast alpha into each element of %ymm1.

Loop: %rdx == x+I and %rcx = n remaining.

Now we've got a vmul**PD** (packed double).

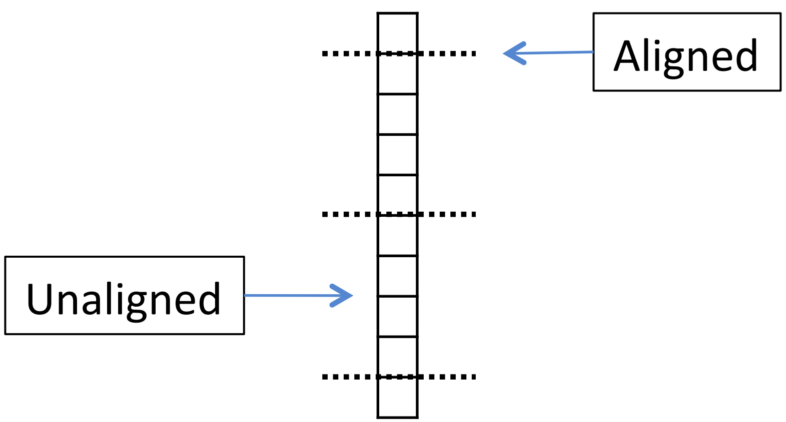
This is called a **split store**. This is better here because the compiler doesn't know if the data is **aligned** to a multiple of 32B.

Sample 1 disassembly (with vectorization)

- Not all compilers (or compiler version!) optimize the same.
- The same compiler optimized differently for different processors (even with the same instructions).
- TL;DR: YMMV. Some variations you may see, even in this small example:
  - **Loop peeling**.
  - Loop unrolling.
  - **Split** loads and stores.
  - **Aligned** vs. **unaligned** loads and stores.
  - Run-time checks to select the algorithm.

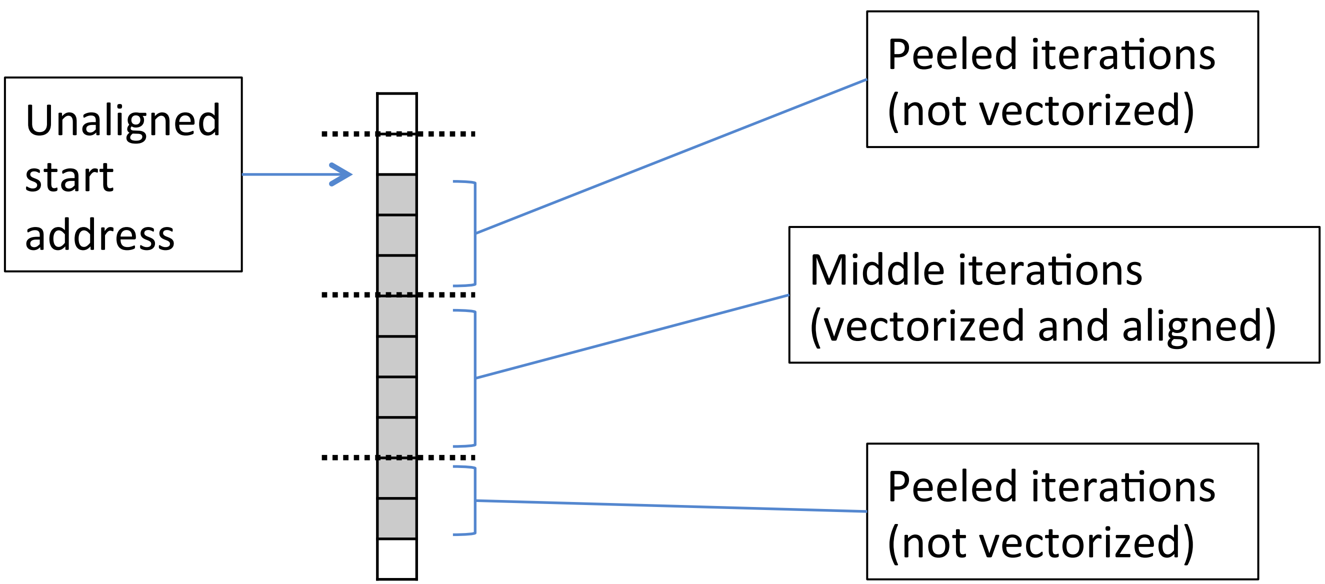
Alignment

- `vmovapd`: Load/store 32B at an address which is a multiple of 32 in bytes.
  - If you give it an address that isn't aligned, you get a segfault.
  - Fast on every architecture.
  - Used to be the only way to load/store vectors.
- `vmovupd`: Load/store 32B of data at any address.
  - May be slower than `vmovapd`, even for aligned data.
  - Not a big deal on newer processors, though (Haswell and later).



Loop peeling

- **Loop peeling** is when the first and last few iterations of a loop are executed separately than the middle portion.
- The compiler may use this to start the middle (vectorized) loop iterations at an **aligned** address.
- Loop peeling is also used to execute the last few loop iterations that fall short of a full vector.



Loop unrolling

- **Loop unrolling** is where multiple iterations of a loop are executed at the same time.
- Loop vectorization is already a form of unrolling, but sometimes the compiler unrolls the vectorized loop as well:

Original loop (vectorized):

```
for
  vmovupd ...
  vmulpd ...
  vmovupd ...
end
```

4x unrolled loop:

```
for
  vmovupd ...
  vmovupd ...
  vmovupd ...
  vmovupd ...
  vmulpd ...
  vmulpd ...
  vmulpd ...
  vmulpd ...
  vmovupd ...
  vmovupd ...
  vmovupd ...
  vmovupd ...
end
```

Sample 2a: DAXPY

Compile with: g++ -O3 -march=native –fno-unroll-loops –c -o sample2a.o sample2a.cxx

```
void daxpy(int n, double alpha,
                const double* x, /* assume incx = 1 */
                      double* y  /* assume incy = 1 */)
{
    for (int i = 0;i < n;i++)
    {
        y[i] += alpha*x[i];
    }
}
```

Sample 2a disassembly

This piece of code checks if **x[0:n]** and **y[0:n]** overlap.

What happens if the arrays **x** and **y** overlap? If they do, writing to **y** will change **x**, so we may have a **loop-carried dependency**.

```
34        leaq      (%rsi,%r11,8), %rcx
38        xorl      %eax, %eax
3a        cmpq      %rdx, %rcx
3d        jb        0x48
3f        leaq      (%rdx,%r11,8), %rcx
43        cmpq      %rsi, %rcx
46        jae       0x9f
```

No overlap

```
60        vmovupd (%rcx), %xmm2
64        vinsertf128     $0x1, 0x10(%rcx), %ymm2, %ymm2
6b        vmulpd  %ymm1, %ymm2, %ymm2
6f        vmovupd (%rax), %xmm3
73        vinsertf128     $0x1, 0x10(%rax), %ymm3, %ymm3
7a        vaddpd  %ymm2, %ymm3, %ymm2
7e        vextractf128    $0x1, %ymm2, 0x10(%rax)
85        vmovupd %xmm2, (%rax)
89        addq    $0x20, %rcx
8d        addq    $0x20, %rax
91        addq    $-0x4, %r11
95        jne     0x60
```

Vectorized version

Overlap

```
b0        vmulsd  (%rcx), %xmm0, %xmm1
b4        vaddsd  (%rdx), %xmm1, %xmm1
b8        vmovsd  %xmm1, (%rdx)
bc        addq    $0x8, %rcx
c0        addq    $0x8, %rdx
c4        decl    %edi
c6        jne     0xb0
```

Scalar version

But I promise they don't overlap!

- If you as the programmer can guarantee that two arrays don't overlap (that is, their pointers don't alias), then you can help the compiler to vectorize:

- Use the **__restrict__** keyword (sometimes **__restrict**) to tell the compiler they don't alias:

```
//
// Use the __restrict__ keyword
// to promise to the compiler
// that x and y don't overlap.
//
const double* __restrict__ x,
        /* assume incx = 1 */
      double* __restrict__ y
       /* assume incy = 1 */)
```

- Explicitly tell the compiler the loo is safe to vectorize:

```
//
// ivdep: ignore assumed data dependencies
//
#pragma GCC ivdep // gcc
#pragma ivdep // icpc
//
// simd: always vectorize
//
#pragma simd //icpc
#pragma omp simd //any compiler with OpenMP 4
for (int i = 0;i < n;i++)
{
    y[i] += alpha*x[i];
}
```

Sample 3a: matrix transpose

Compile with: g++ -O3 -march=native –fno-unroll-loops –c -o sample3a.o sample3a.cxx

```
// transpose the column-major m*n matrix A
// into the column-major n*m matrix B
void transpose(int m, int n,
               const double* __restrict__ A, int lda,
                     double* __restrict__ B, int ldb)
{
    for (int i = 0;i < m;i++)
    {
        for (int j = 0;j < n;j++)
        {
            B[i*ldb + j] = A[i + j*lda];
        }
    }
}
```

Sample 3a disassembly

```
20    movl    %esi, %r14d
23    movq    %rdx, %rcx
26    movq    %r8, %rbx
29    nopl    (%rax)
30    movq    (%rcx), %rax
33    movq    %rax, (%rbx)
36    addq    $0x8, %rbx
3a    addq    %r11, %rcx
3d    decl    %r14d
40    jne     0x30
42    incq    %r10
45    addq    %r9, %r8
48    addq    $0x8, %rdx
4c    cmpl    %edi, %r10d
4f    jne     0x20
```

- No “mul” anywhere… the compiler has applied strength reduction for us.
- Not vectorized, but it doesn’t even use the floating-point registers!

Sample 3b: unrolled matrix transpose

- The compiler can't vectorize the inner loop because A is accessed in increments of lda. But, we can unroll both loops to get contiguous access in both matrices:

```
// assume m%4 == 0 and n%4 == 0
for (int i = 0;i < m;i += 4)
{
    for (int j = 0;j < n;j += 4)
    {
        const double* __restrict__ Asub = &A[i + j*lda];
              double* __restrict__ Bsub = &B[i*ldb + j];

         Bsub[0*ldb + 0] = Asub[0 + 0*lda];   Bsub[1*ldb + 0] = Asub[1 + 0*lda];
         Bsub[0*ldb + 1] = Asub[0 + 1*lda];   Bsub[1*ldb + 1] = Asub[1 + 1*lda];
         Bsub[0*ldb + 2] = Asub[0 + 2*lda];   Bsub[1*ldb + 2] = Asub[1 + 2*lda];
         Bsub[0*ldb + 3] = Asub[0 + 3*lda];   Bsub[1*ldb + 3] = Asub[1 + 3*lda];

         Bsub[2*ldb + 0] = Asub[2 + 0*lda];   Bsub[3*ldb + 0] = Asub[3 + 0*lda];
         Bsub[2*ldb + 1] = Asub[2 + 1*lda];   Bsub[3*ldb + 1] = Asub[3 + 1*lda];
         Bsub[2*ldb + 2] = Asub[2 + 2*lda];   Bsub[3*ldb + 2] = Asub[3 + 2*lda];
         Bsub[2*ldb + 3] = Asub[2 + 3*lda];   Bsub[3*ldb + 3] = Asub[3 + 3*lda];
    }
}
```

Sample 3b disassembly

```
1ff     movq    -0x48(%rbp), %rax
203     movq    %rdx, (%r14,%rax,8)
207     movq    0x8(%r11,%rsi,8), %rdx
20c     movq    %rdx, (%r14,%r12,8)
210     movq    -0x88(%rbp), %rax
217     movq    (%r11,%rax,8), %rdx
21b     movq    -0x40(%rbp), %rax
21f     movq    %rdx, (%r14,%rax,8)
223     movq    0x10(%r11), %rdx
227     movq    %rdx, (%r14,%r15,8)
22b     movq    -0x80(%rbp), %rax
22f     movq    (%r11,%rax,8), %rdx
233     movq    %rdx, 0x8(%r14,%r15,8)
238     movq    -0x78(%rbp), %rax
23c     movq    (%r11,%rax,8), %rdx
```

No vector instructions anywhere!

We just get 16 copies of what we had before.

And so on…

Manual vectorization

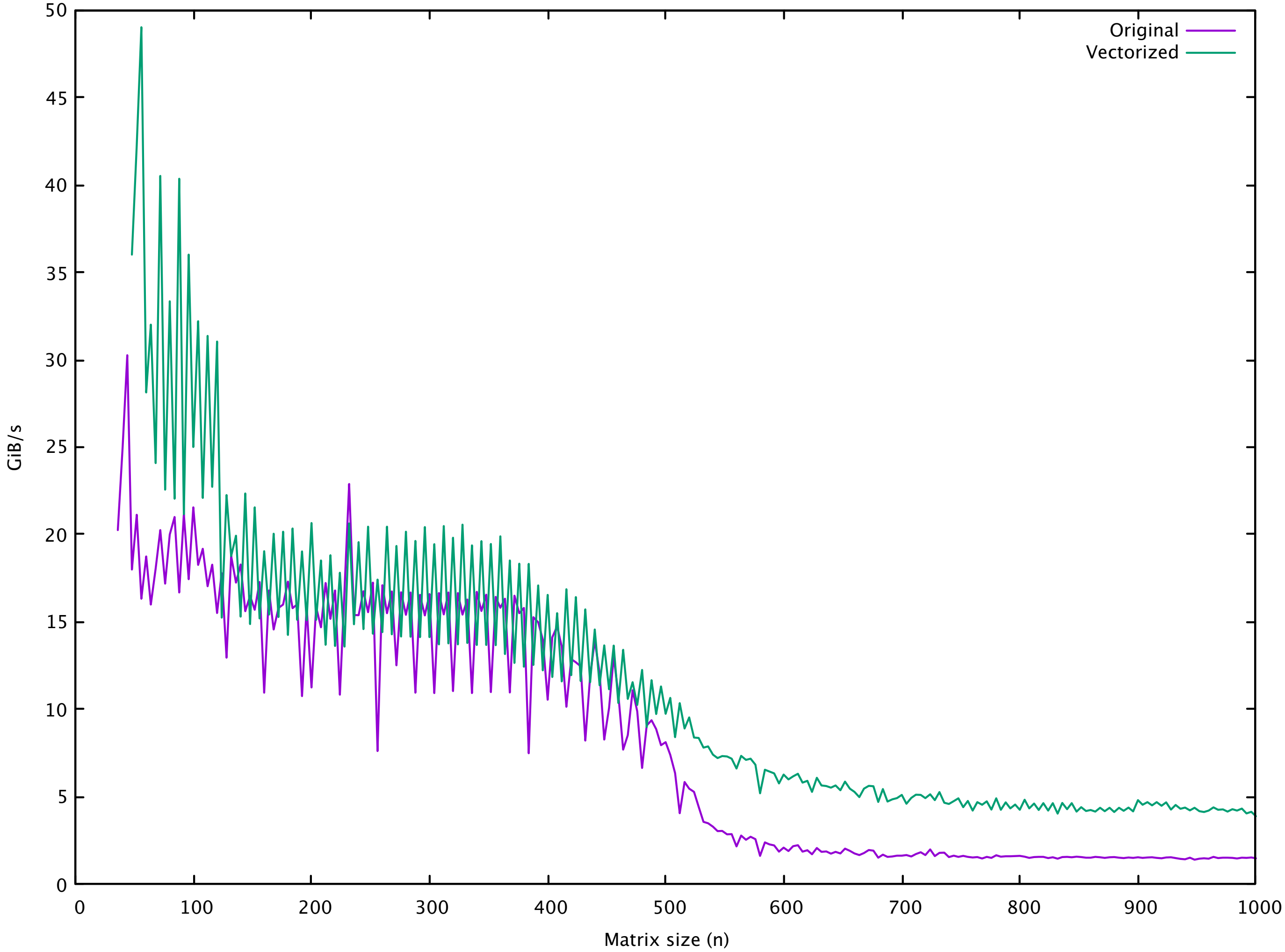
- Since the compiler can’t vectorize this **block** itself, we will need to do the vectorization ourselves.
- We can always put **inline assembly** into the code, but this is tedious and can prevent some other compiler optimizations.
  - But some times it is the only way.
- Instead, we can use **compiler intrinsics** to do the vectorization.
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide is your bible for this

Sample 3c: manually vectorized matrix transpose

```
__m256d Areg[4], Breg[4];

// assume m%4 == 0 and n%4 == 0
for (int i = 0;i < m;i += 4)
{
    for (int j = 0;j < n;j += 4)
    {
        const double* __restrict__ Asub = &A[i + j*lda];
              double* __restrict__ Bsub = &B[i*ldb + j];

         Areg[0] = _mm256_loadu_pd(Asub + 0*lda);
         Areg[1] = _mm256_loadu_pd(Asub + 1*lda);
         Areg[2] = _mm256_loadu_pd(Asub + 2*lda);
         Areg[3] = _mm256_loadu_pd(Asub + 3*lda);

         transpose_4x4(Areg, Breg);

         _mm256_storeu_pd(Bsub + 0*ldb, Breg[0]);
         _mm256_storeu_pd(Bsub + 1*ldb, Breg[1]);
         _mm256_storeu_pd(Bsub + 2*ldb, Breg[2]);
         _mm256_storeu_pd(Bsub + 3*ldb, Breg[3]);
    }
}
```

```
void transpose_4x4(__m256d A[4], __m256d B[4])
{
    __m256d tmp[4];
    // A[0] = (A00, A10, A20, A30)
    // A[1] = (A01, A11, A21, A31)
    // A[2] = (A02, A12, A22, A32)
    // A[3] = (A03, A13, A23, A33)
    tmp[0] = _mm256_shuffle_pd(A[0], A[1], 0x0);
    tmp[1] = _mm256_shuffle_pd(A[0], A[1], 0xf);
    tmp[2] = _mm256_shuffle_pd(A[2], A[3], 0x0);
    tmp[3] = _mm256_shuffle_pd(A[2], A[3], 0xf);
    // tmp[0] = (A00, A01, A20, A21)
    // tmp[1] = (A10, A11, A30, A31)
    // tmp[2] = (A02, A03, A22, A23)
    // tmp[3] = (A12, A13, A32, A33)
    B[0] = _mm256_permute2f128_pd(tmp[0], tmp[2], 0x20);
    B[1] = _mm256_permute2f128_pd(tmp[1], tmp[3], 0x20);
    B[2] = _mm256_permute2f128_pd(tmp[0], tmp[2], 0x31);
    B[3] = _mm256_permute2f128_pd(tmp[1], tmp[3], 0x31);
    // B[0] = (A00, A01, A02, A03)
    // B[1] = (A10, A11, A12, A13)
    // B[2] = (A20, A21, A22, A23)
    // B[3] = (A30, A31, A32, A33)
}
```

Why bother?



Original
Vectorized
GiB/s
Matrix size (n)
0
5
10
15
20
25
30
35
40
45
50
0
100
200
300
400
500
600
700
800
900
1000

Questions?