Code Profiling and Optimization

Day 09

Explicit Vectorization

Subhrajit & Pratyush

Recap

Earlier we saw, the compiler doing the auto-vectorization

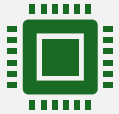
We saw a report while compiling using –fopt-info-vec

But how exactly does it differ?


Assembly Dumps!

Getting Started with Explicit AVX

To save us writing in Assembly, Intel has a set of APIs (Called as Intrinsics!)



<immintrin.h> in Intel CPUs have those APIs

And while compiling use –mavx512f (most of the times)

Guide: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Sum of two arrays(AVX512)

Suppose we need to add two integer arrays of size 32 elements

- int arr1[32] and arr2[32], size of(int) -> 4 Bytes.

If we want to load it in 512-bit vectors, each 512-vector can accommodate 16 integers at a time

So, we would two pass of 512-bit vector operations

But how to load to a 512-bit vector?

Load onto AVX register

load_epi32

```
__m128i _mm_load_epi32 (void const* mem_addr)                                  vmovdqa32
__m128i _mm_mask_load_epi32 (__m128i src, __mmask8 k, void const* mem_addr)    vmovdqa32
__m128i _mm_maskz_load_epi32 (__mmask8 k, void const* mem_addr)                vmovdqa32
__m256i _mm256_load_epi32 (void const* mem_addr)                               vmovdqa32
__m256i _mm256_mask_load_epi32 (__m256i src, __mmask8 k, void const* mem_addr) vmovdqa32
__m256i _mm256_maskz_load_epi32 (__mmask8 k, void const* mem_addr)             vmovdqa32
__m512i _mm512_load_epi32 (void const* mem_addr)                               vmovdqa32
__m512i _mm512_mask_load_epi32 (__m512i src, __mmask16 k, void const* mem_addr) vmovdqa32
__m512i _mm512_maskz_load_epi32 (__mmask16 k, void const* mem_addr)            vmovdqa32
```

Load 32-bits onto AVX512 register

```
// Sample AVX512 intrinsics code to compute sum of two arrays

// arr1[] and arr2[] are two input arrays of size n


for (int i = 0; i < n; i += 16) {
    // Load 16 elements from arr1[] and arr2[]
    __m512i v1 = _mm512_load_epi32(&arr1[i]);
    __m512i v2 = _mm512_load_epi32(&arr2[i]);

    // Other operations
    // ...
}
```

```
// Note: AVX loads are contiguous in nature.
// Example: _mm512_load_epi32(&arr1[i]) loads 16 elements starting from arr1[i] to arr1[i+15].
```

Compute Sum of two vectors

_mm512_add

`__m512i _mm512_add_epi16 (__m512i a, __m512i b)`	vpaddw
`__m512i _mm512_add_epi32 (__m512i a, __m512i b)`	vpaddd
`__m512i _mm512_add_epi64 (__m512i a, __m512i b)`	vpaddq
`__m512i _mm512_add_epi8 (__m512i a, __m512i b)`	vpaddb
`__m512d _mm512_add_pd (__m512d a, __m512d b)`	vaddpd
`__m512h _mm512_add_ph (__m512h a, __m512h b)`	vaddph
`__m512 _mm512_add_ps (__m512 a, __m512 b)`	vaddps
`__m512d _mm512_add_round_pd (__m512d a, __m512d b, int rounding)`	vaddpd
`__m512h _mm512_add_round_ph (__m512h a, __m512h b, int rounding)`	vaddph
`__m512 _mm512_add_round_ps (__m512 a, __m512 b, int rounding)`	vaddps
`__m512i _mm512_adds_epi16 (__m512i a, __m512i b)`	vpaddsw
`__m512i _mm512_adds_epi8 (__m512i a, __m512i b)`	vpaddsb
`__m512i _mm512_adds_epu16 (__m512i a, __m512i b)`	vpaddusw
`__m512i _mm512_adds_epu8 (__m512i a, __m512i b)`	vpaddusb

Compute the Sum of two 512-vectors (32-bit data inside)

```
// Sample AVX512 intrinsics code to compute sum of two arrays

// arr1[] and arr2[] are two input arrays of size n


for (int i = 0; i < n; i += 16) {
    // Load 16 elements from arr1[] and arr2[]
    __m512i v1 = _mm512_load_epi32(&arr1[i]);
    __m512i v2 = _mm512_load_epi32(&arr2[i]);

    // add v1 and v2
    __m512i result = _mm512_add_epi32(v1, v2);

    // other operations
    // ...
}
```

```
// Note: Here we're using _mm512_add_epi32() to add two vectors.
// And, using epi32 because we're adding 32-bit integers.
```

Store the 512 results in memory

_mm512_store

void _mm512_store_epi32 (void* mem_addr, __m512i a)	vmovdqa32
void _mm512_store_epi64 (void* mem_addr, __m512i a)	vmovdqa64
void _mm512_store_pd (void* mem_addr, __m512d a)	vmovapd
void _mm512_store_ph (void * mem_addr, __m512h a)	vmovaps
void _mm512_store_ps (void* mem_addr, __m512 a)	vmovaps
void _mm512_store_si512 (void* mem_addr, __m512i a)	vmovdqa32
void _mm512_storeu_epi16 (void* mem_addr, __m512i a)	vmovdqu16
void _mm512_storeu_epi32 (void* mem_addr, __m512i a)	vmovdqu32
void _mm512_storeu_epi64 (void* mem_addr, __m512i a)	vmovdqu64
void _mm512_storeu_epi8 (void* mem_addr, __m512i a)	vmovdqu8
void _mm512_storeu_pd (void* mem_addr, __m512d a)	vmovupd
void _mm512_storeu_ph (void * mem_addr, __m512h a)	vmovups
void _mm512_storeu_ps (void* mem_addr, __m512 a)	vmovups
void _mm512_storeu_si512 (void* mem_addr, __m512i a)	vmovdqu32

Store the 512-bit result (32-bit sums) in memory

```
// Sample AVX512 intrinsics code to compute sum of two arrays

// arr1[] and arr2[] are two input arrays of size n


for (int i = 0; i < n; i += 16) {
    // Load 16 elements from arr1[] and arr2[]
    __m512i v1 = _mm512_load_epi32(&arr1[i]);
    __m512i v2 = _mm512_load_epi32(&arr2[i]);

    // add v1 and v2
    __m512i result = _mm512_add_epi32(v1, v2);

    // Store the result back to arr1[]
    _mm512_store_epi32(&arr1[i], result);
}

// Note: _mm512_store_epi32() stores 16 elements starting from arr1[i] to arr1[i+15].
```

Important things to keep in mind!

- Loads/Stores are contiguous (row-major) in nature
- Make sure memory is properly aligned
  - E.g. If your array arr[0…n-1], and if try to load from arr[2] it will give segfaults
- If aligned load/stores are not possible, then use unaligned load/stores, but would typically incur more cycles
- All of the computations that we perform on the 512-bit registers, are localized within lanes only!

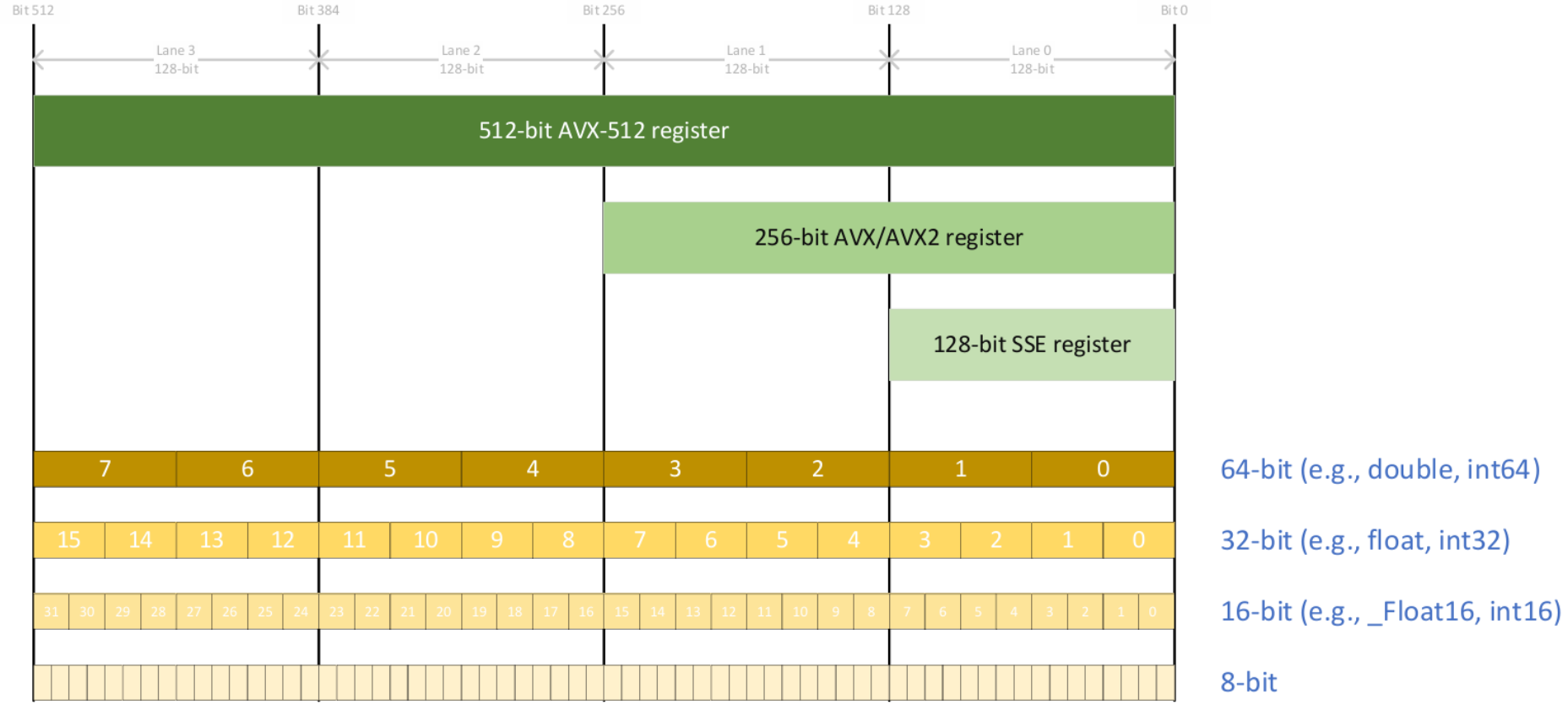


Figure 1. Layout of Various Sizes of SIMD Register and How Each Can Be Broken Down into Smaller Subgroups of Elements

Lanes

- All of the computations that we perform on the 512-bit registers, are localized within lanes only!
- Newer Intel intrinsics like mavx512vbmi2 have support for across lanes, but would typically require a set of intrinsics to be used sequentially.

Well, that’s all for a kick-start!

Summary of SC-369

Compiler Flags	O0,O1,O2,O3,Ofast,g
Debugging Tools	GDB, OBJDUMP, VALGRIND
Measurement Tools	Timing tools(RUSAGE, Timespec, RDTSC) Performance Tools (Perf – terminal utility, event open)
Memory Layout	Segments of overall memory
Multi-threading	Posix threads, OpenMP
Vectorization	Auto-vectorization, Explicit Vectorization using intrinsics (AVX)

One bonus!

- Compiler hint for branching misses
- Hint the compiler using LIKELY and UNLIKELY constructs!
- Example:

```
int naive_function(int *a, int *b, int *c, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        float r = (float)rand() / RAND_MAX;
        if (r < 0.01)
        {
            c[i] = a[i] * b[i];
        }
        else
        {
            c[i] = a[i] + b[i];
        }
    }
    return 0;
}
```

One bonus!

```
#define LIKELY(x) __builtin_expect(!!(x), 1)
#define UNLIKELY(x) __builtin_expect(!!(x), 0)
```

- Hint the compiler using LIKELY and UNLIKELY constructs!
- Example:

```
int naive_function(int *a, int *b, int *c, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        float r = (float)rand() / RAND_MAX;
        if (r < 0.01)
        {
            c[i] = a[i] * b[i];
        }
        else
        {
            c[i] = a[i] + b[i];
        }
    }
    return 0;
}
```

```
int optimized_function(int *a, int *b, int *c, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        float r = (float)rand() / RAND_MAX;
        if (UNLIKELY(r < 0.01))
        {
            c[i] = a[i] * b[i];
        }
        else
        {
            c[i] = a[i] + b[i];
        }
    }
    return 0;
}
```


See you tomorrow for the Challenge!