# COMP3320/6464: Loop vectorization Streaming SIMD Extensions

Jiri Jaros, Alistair Rendell and Jun Zhou



## 14.1 Exploiting Parallelism

We can exploit parallelism in two ways:

1. **At the core level**
   - Rewrite the code to parallelize operations across many cores (pthreads, MPI)
   - Make use of extensions to the programming language (OpenMP)
2. **At the instruction level**
   - Out-of-order execution and pipelining
   - Single Instruction Multiple Data (SIMD)



## 14.2 Vector or SIMD units

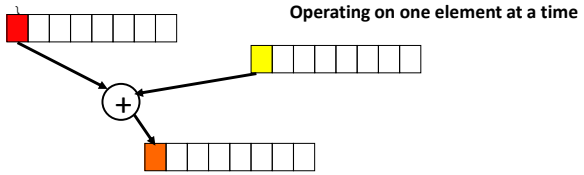
- A vector or SIMD (Single Instruction Multiple Data) Unit is a unit used to accelerate loops
- It exists in almost all general purpose processors:
  - INTEL SSE: Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture
  - AltiVeC is a floating point and integer instruction set designed and owned by Appple, IBM and Freescale Semiconductor.
- A similar architecture is found in game consoles (PS2, PS3) and GPUs (NVIDIA's GeForce)



## 14.3 Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

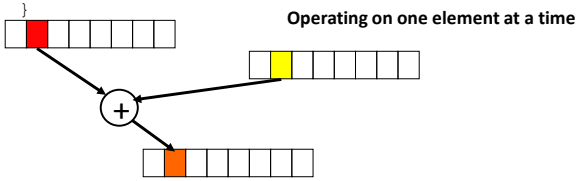
```
void
array_add(int A[], int B[], int C[], int length)
  {
  int i;
  for (i = 0 ; i < length ; i++) {
   C[i] = A[i] + B[i];
  }
}
```



## 14.4 Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length)
  {
  int i;
  for (i = 0 ; i < length ; i++) {
   C[i] = A[i] + B[i];
  }
}
```

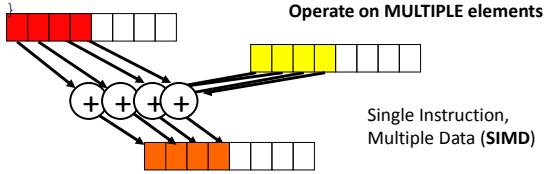





## 14.5 Exploiting Parallelism at the Instruction level (SIMD)

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length)
  {
  int i;
  for (i = 0 ; i < length ; i++) {
   C[i] = A[i] + B[i];
  }
}
```





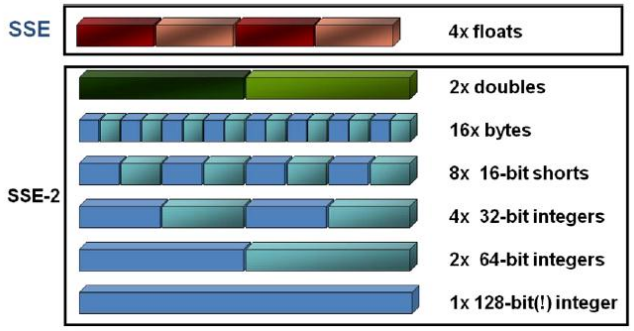

## 14.6 SIMD Extensions

- **SSE** (1999) brings 70 new instructions and improves obsolete MMX
- **SSE2** (2000) introduces other 144 instructions. It simplifies compiler support for easy coding. Supports doubles and 32b integers
- **SSE3** (2005) adds 13 new instructions for multi-thread support and HyperThreading
- **SSE4** (2007) adds 54 new instructions not designated for multimedia (text processing, strings, fixed point arithmetic)
- **AVX** (2008) extends vector register to 256b (up 1024 in future)



## 14.7 Intel SSE/SSE2 as an example of SIMD

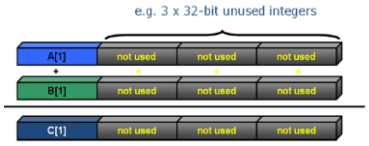
- Added new 128 bit registers (XMM0 – XMM7), each can store



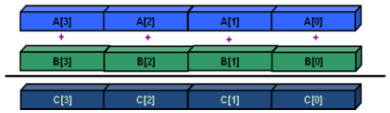
## 14.8 Simple SSE/SSE2 example

```
for (i = 0; i < MAX; i++){
   c[i] = a[i] + b[i]
}
```

No vectorization




Vectorized






## 14.9 Compiler autovectorization

- **Loop unrolling combined with the generation of packed SIMD instructions**
- **GCC enables with –O3, intel with -O2**
  - Instruction set specified by `-msse2` ( `-msse4.1`)
- **Reports from vectorization process**
  `-ftree-vectorizer-verbose=5` (gcc)
  `-vec-report5` (intel)



## 14.10 What sort of loops can be vectorized?

1. **Countable** – The loop trip count must be known at entry to the loop at runtime.
2. **Single entry and single exit** – no break
3. **Straight-line code** – it is not possible for different iterations to have different flow control (must not branch). If statements allowed if they can be implemented as masked assignments.
4. **The innermost loop of a nest** – possible loop interchange in previous optimization phases
5. **No function calls** – some intrinsic math functions allowed (sin, log, pow,…)



## 14.11 Obstacles to vectorization

1. **Non-contiguous memory accesses** – four consecutive floats (ints) can be loaded directly. If there is a stride, they have to be loaded separately using multiple instructions.
2. **Not aligned data structures** – may results in multiple load instructions.
3. **Data dependencies** – (RAW, WAR, WAW), can threat reductions.

## 14.12 Intel C compiler vectorization pragmas

- **#pragma ivdep** may be used to tell the compiler that it may safely ignore any potential data dependencies. (The compiler will not ignore proven dependencies). Use of this pragma when there are in fact dependencies may lead to incorrect results.
- **#pragma loop count (n)** may be used to advise the compiler of the typical trip count of the loop. This may help the compiler to decide whether vectorization is worthwhile, or whether or not it should generate alternative code paths for the loop.
- **#pragma vector always** asks the compiler to vectorize the loop if it is safe to do so, whether or not the compiler thinks that will improve performance.
- **#pragma vector align** asserts that data within the following loop is aligned (to a 16 byte boundary, for SSE instruction sets).
- **#pragma novector** asks the compiler not to vectorize a particular loop
- **#pragma vector nontemporal** gives a hint to the compiler that data will not be reused, and therefore to use streaming stores that bypass cache.



## 14.13 Streaming SIMD data types

| New data type | Content | SSE extension | SSE2 extension | SSE4 extension |
|---|---|---|---|---|
| `__m128` | 4 x float | Available | Available | Available |
| `__m128d` | 2 x double | Not available | Available | Available |
| `__m128i` | 16 x char<br>8 x short<br>4 x int | Not available | Available | Available |



## 14.14 SSE instructions: Naming and usage syntax

- Most intrinsic names use the following notational convention:

```
_mm_<intrin_op>_<suffix>
```

- **<intrin_op>** indicates the basic operation of the intrinsic; for example, `add` for addition and `sub` for subtraction.
- **<suffix>** denotes the type of data the instruction operates on.
  - The first one or two letters of each suffix denote whether the data is
    - p packed
    - ep extended packed
    - S scalar
  - The remaining letters and numbers denote the type, with notation as follows:
    - s single-precision floating point
    - d double-precision floating point
    - i32 signed 32-bit integer
    - u32 unsigned 32-bit integer
    - …



## 14.15 SSE instructions: Memory access

- Load operations

```
__m128 _mm_load1_ps(float * p)
```

  - Load one SP FP value into all four words

| R0 | R1 | R2 | R3 |
|---|---|---|---|
| *p | *p | *p | *p |

```
__m128 _mm_load_ps(float * p )
```

  - Load four SP FP values, address aligned

| R0 | R1 | R2 | R3 |
|---|---|---|---|
| p[0] | p[1] | p[2] | p[3] |

## 14.16 SSE instructions: Memory access

- Set operations

`__m128 _mm_set1_ps(float w)`

- Sets the four SP FP values to w

| R0 | R1 | R2 | R3 |
|---|---|---|---|
| w | w | w | w |

`__m128 _mm_set_ps( float z, float y, float x, float w )`

- Sets the four SP FP values to the four inputs.

| R0 | R1 | R2 | R3 |
|---|---|---|---|
| w | x | y | z |

## 14.17 SSE instructions: Memory access

- Store operations

`void _mm_store1_ps(float * p, __m128 a )`

- Stores the lower SP FP value across four words.

| p[0] | p[1] | p[2] | p[3] |
|---|---|---|---|
| a0 | a0 | a0 | a0 |

`void _mm_store_ps(float *p, __m128 a)`

- Stores the four SP FP values. The address must be 16-byte-aligned

| p[0] | p[1] | p[2] | p[3] |
|---|---|---|---|
| a0 | a1 | a2 | a3 |

## 14.18 SSE instructions: Arithmetic operations

`__m128 _mm_add_ss(__m128 a, __m128 b)`

- Adds the lower single-precision, floating-point (SP FP) values of a and b; the upper 3 SP FP values are passed through from a.

| p[0] | p[1] | p[2] | p[3] |
|---|---|---|---|
| a0 + b0 | a1 | a2 | a3 |

`__m128 _mm_add_ps(__m128 a, __m128 b)`

- Stores four SP FP values. The address must be 16-byte-aligned

| p[0] | p[1] | p[2] | p[3] |
|---|---|---|---|
| a0 + b0 | a1 + b1 | a2 + b2 | a3 + b3 |

## 14.19 Vector addition example

**Basic Implementation**

```
void VectorAdd(float *a, float *b, float *c, size_t size){

    for (size_t i = 0; i < size; i++){
        c[i] = a[i] + b[i];
    }
}
```

**Loop unrolling**

```
void VectorAddUnrolled(float *a, float *b, float *c, size_t size){
    for (size_t i = 0; i < (size/4) * 4; i+=4){
        c[i]   = a[i]   + b[i];
        c[i+1] = a[i+1] + b[i+1];
        c[i+2] = a[i+2] + b[i+2];
        c[i+3] = a[i+3] + b[i+3];
    }

    for (size_t i = (size/4) * 4; i < size; i++){
        c[i] = a[i] + b[i];
    }
}
```

## 14.20 Vector addition example

### SSE2 implementation

```
void VectorAddSSE(  float *a, float *b, float *c,
                    size_t size){

    for (size_t i = 0; i < (size/4) * 4; i+=4){
      __m128 sse_a = _mm_load_ps(&a[i]);
      __m128 sse_b = _mm_load_ps(&b[i]);

      __m128 sse_c = _mm_add_ps(sse_a,sse_b);

                     _mm_store_ps(&c[i],sse_c);

    }

    for (size_t i = (size/4) * 4; i < size; i++){
        c[i] = a[i] + b[i];
    }
}
```



## 14.21 Vector dot product example

### Basic Implementation

```
void VectorDot(    float *a, float *b, float *result,
                   size_t size){

    float sum = 0.0f;

    for (size_t i = 0; i < size; i++){
        sum += a[i] * b[i];
    }

    *result = sum;
}
```



## 14.22 Vector dot product example

### Loop unrolling

```
void VectorDotUnrolled(float *a, float *b, float *result, size_t
size){

    float sum[4] = {0.0f, 0.0f, 0.0f, 0.0f};
    for (size_t i = 0; i < (size/4) * 4; i+=4){
        sum[0] += a[i]   * b[i];
        sum[1] += a[i+1] * b[i+1];
        sum[2] += a[i+2] * b[i+2];
        sum[3] += a[i+3] * b[i+3];
    }

    for (size_t i = (size/4) * 4; i < size; i++){
        sum[0] += a[i] * b[i];
    }

    *result = sum[0] + sum[1] + sum[2] + sum[3];
}
```



## 14.23 Vector dot product example

### SSE2 implementation

```
void VectorDotSSE(float *a, float *b, float *result, size_t size){

    float sum[4] __attribute__ ((aligned (16))) = {0.0f, 0.0f, 0.0f, 0.0f};

    __m128 sse_sum = _mm_set1_ps(0.0f);

    for (size_t i = 0; i < (size/4) * 4; i+=4){
      __m128 sse_a   = _mm_load_ps(&a[i]);
      __m128 sse_b   = _mm_load_ps(&b[i]);

      __m128 sse_tmp = _mm_mul_ps(sse_a  ,sse_b);
             sse_sum = _mm_add_ps(sse_tmp,sse_sum);
    }

    _mm_store_ps(sum, sse_sum);


    for (size_t i = (size/4) * 4; i < size; i++){
        sum[0] += a[i] * b[i];
    }
    *result = sum[0] + sum[1] + sum[2] + sum[3];
}
```

## 14.24 Conclusions

- Streaming SIMD can significantly accelerate execution of loop
- In simple cases, the compiler is able to vectorize the code by itself
- Modern also compilers provide SSE optimized math libraries
- In a general case, it's up to a programmer to vectorize the code using intrinsic instructions