HPC Practical Course

2.1: SIMD with Headers

V. Akishina, I. Kisel, G. Kozlov, I. Kulakov, M. Pugach, M. Zyzak
Goethe University of Frankfurt am Main
2015
Gordon Earle Moore (born January 3, 1929) is an American businessman and co-founder and Chairman Emeritus of Intel Corporation and the author of Moore's law. As of January 2015, his net worth is $6.7 billion.

In 1965, Gordon E. Moore was working as the director of research and development (R&D) at Fairchild Semiconductor. He was asked by Electronics Magazine to predict what was going to happen in the semiconductor components industry over the next ten years. In an article published on April 19, 1965, Moore observed that the number of components (transistors, resistors, diodes or capacitors) in a dense integrated circuit had doubled approximately every year, and speculated that it would continue to do so for at least the next ten years. In 1975, he revised the forecast rate to approximately every two years. Carver Mead popularized the phrase “Moore’s law”. The prediction has become a target for miniaturization in the semiconductor industry, and has had widespread impact in many areas of technological change.

Moore's Law:
- the number of transistors doubles approximately every two years
- surprisingly good prediction
- but it “only” tells about the number of transistors
- the clock rate mostly fixed

Performance increase by means of:
- improved parallelism
- faster and larger memory caches
Michael J. Flynn (born May 20, 1934) is an American professor emeritus at Stanford University. Michael Flynn classified in 1966 multi-processor architectures into four categories (Flynn, M.J. (1972). "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput. C-21: 948.). The classification system has stuck, and has been used as a tool in design of modern processors and their functionalities. Since the rise of multiprocessing CPUs, a multiprogramming context has evolved as an extension of the classification system.

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

- **SISD** - a single processor executes a single operation on data stored in one memory location.
- **SIMD** - multiple processors perform the same operation on data stored in multiple memory locations.
- **MISD** - multiple processors perform multiple operations on data stored in one memory location.
- **MIMD** - multiple processors perform multiple operations on data stored in multiple memory locations.

**CPUs and GPUs**

Multi-core CPUs are MIMD machines. An important sub-category of the MIMD class is:

- **SPMD** - single program multiple data - multiple cooperating processes execute one program.

GPUs are SPMD machines. SPMD and SIMD are not mutually exclusive. A SIMD machine executes in lockstep on different data, while an SPMD machine executes the same program on multiple data, but not necessarily in lockstep.
Cores and Threads realize the task level of parallelism

**Some remarks:**
- hardware parallelism for free;
- many-core machine is not a batch farm;
- use core/thread parallelism at the event level;
- scalar code is useless;
- use vector programming;

\[ \text{Speed-up} = (1 + \text{HT}) \times N \text{ cores} \times N \text{ sockets} \times \text{SIMD width} \]

\[ S_{\text{lxir075@gsi}} = (1 + 0.3) \times 10 \times 4 \times 4 = 200 \]

Vectors (SIMD) = data level of parallelism

**SIMD** = Single Instruction, Multiple Data
Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn’s taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use.

An application that may take advantage of SIMD is one where the same value is being added to (or subtracted from) a large number of data points, a common operation in many multimedia applications. With a SIMD processor there are two improvements to this process. For one the data is understood to be in blocks, and a number of values can be loaded all at once. Another advantage is that SIMD systems typically include only those instructions that can be applied to all of the data in one operation. In other words, if the SIMD system works by loading up eight data points at once, the add operation being applied to the data will happen to all eight values at the same time. Although the same is true for any super-scalar processor design, the level of parallelism in a SIMD system is typically much higher.

Disadvantages:

- Not all algorithms can be vectorized easily.
- It also has large register files which increases power consumption and chip area.
- Currently, implementing an algorithm with SIMD instructions usually requires human labor; auto-vectorization in compilers is an active area of computer science research.
- Programming with particular SIMD instruction sets can involve numerous low-level challenges.
SIMD Registers

SIMD register:

128 bit

__m128

double  double
float   float   float   float

• There are CPUs with 256 bit and 512 bit registers

Speed up of factor 4
Basic Single Precision SSE SIMD Instructions

Arithmetic:

- \_mm\_add\_ps (Va, Vb) \quad a + b
- \_mm\_sub\_ps (Va, Vb) \quad a - b
- \_mm\_mul\_ps (Va, Vb) \quad a \times b
- \_mm\_div\_ps (Va, Vb) \quad a / b

Logical:

- \_mm\_and\_ps (Va, Vb) \quad a \& b
- \_mm\_or\_ps (Va, Vb) \quad a | b
- \_mm\_xor\_ps (Va, Vb) \quad a ^ b

Comparison:

- \_mm\_cmplt\_ps (Va, Vb) \quad a < b
- \_mm\_cmpeq\_ps (Va, Vb) \quad a <= b
- \_mm\_cmpgt\_ps (Va, Vb) \quad a > b
- \_mm\_cmpge\_ps (Va, Vb) \quad a >= b
- \_mm\_cmpeq\_ps (Va, Vb) \quad a == b

Extra:

- \_mm\_min\_ps (Va, Vb) \quad \text{min} \ (a, b)
- \_mm\_max\_ps (Va, Vb) \quad \text{max} \ (a, b)
- \_mm\_rcp\_ps (Va) \quad 1/a
- \_mm\_sqrt\_ps (Va) \quad \sqrt{a}
- \_mm\_rsqrt\_ps (Va) \quad 1/\sqrt{a}

There are much more instructions, including instructions for SIMD vectors with double precision values.
The instructions can be wrapped by C++ class, which overloads standard operators (+, -, *, \, >, etc.) for the convenience of user.

```
class F32vec4
{
    public:
        __m128 v;

        float & operator[]( int i ) { return (reinterpret_cast<float*>(&v))[i]; }
        float operator[]( int i ) const { return (reinterpret_cast<const float*>(&v))[i]; }

        F32vec4() :v(_mm_set_ps1(0)){};
        F32vec4( const __m128 &a ) :v(a) {};
        F32vec4( const float &a ) :v(_mm_set_ps1(a)) {};

        F32vec4( const float &f0, const float &f1, const float &f2, const float &f3 ) :v(_mm_set_ps(f3,f2,f1,f0)) {};

        /* Arithmetic Operators */
        friend F32vec4 operator +(const F32vec4 &a, const F32vec4 &b) { return _mm_add_ps(a,b); }

        /* Square Root */
        friend F32vec4 sqrt ( const F32vec4 &a ) { return _mm_sqrt_ps(a); }

        __attribute__((aligned(16)));

typedef F32vec4 fvec;
typedef float fscal;
const int fvecLen = 4;
```
typedef F32vec4 Fvec_t;

/* Arithmetic Operators */
friend F32vec4 operator + ( const F32vec4 &a, const F32vec4 &b ) { return _mm_add_ps(a, b); }
friend F32vec4 operator -( const F32vec4 &a, const F32vec4 &b ) { return _mm_sub_ps(a, b); }
friend F32vec4 operator *( const F32vec4 &a, const F32vec4 &b ) { return _mm_mul_ps(a, b); }
friend F32vec4 operator /( const F32vec4 &a, const F32vec4 &b ) { return _mm_div_ps(a, b); }

/* Functions */
friend F32vec4 min( const F32vec4 &a, const F32vec4 &b ) { return _mm_min_ps(a, b); }
friend F32vec4 max( const F32vec4 &a, const F32vec4 &b ) { return _mm_max_ps(a, b); }

/* Square Root */
friend F32vec4 sqrt( const F32vec4 &a ) { return _mm_sqrt_ps(a); }

/* Absolute value */
friend F32vec4 fabs( const F32vec4 &a ) { return _mm_and_ps(a, _f32vec4_abs_mask); }

/* Logical */
friend F32vec4 operator&( const F32vec4 &a, const F32vec4 &b ) { // mask returned
    return _mm_and_ps(a, b); }
friend F32vec4 operator|( const F32vec4 &a, const F32vec4 &b ) { // mask returned
    return _mm_or_ps(a, b); }
friend F32vec4 operator^( const F32vec4 &a, const F32vec4 &b ) { // mask returned
    return _mm_xor_ps(a, b); }
friend F32vec4 operator!( const F32vec4 &a ) { // mask returned
    return _mm_xor_ps(a, _f32vec4_true); }
friend F32vec4 operator||( const F32vec4 &a, const F32vec4 &b ) { // mask returned
    return _mm_or_ps(a, b); }

/* Comparison */
friend F32vec4 operator<( const F32vec4 &a, const F32vec4 &b ) { // mask returned
    return _mm_cmplt_ps(a, b); }

SIMD instructions
inline void AddMaterial( TrackV &track, Station &st, Fvec_t &qp0 )
{
    cnst mass2 = 0.1396*0.1396;

    Fvec_t tx = track.T[2];
    Fvec_t ty = track.T[3];
    Fvec_t txtx = tx*tx;
    Fvec_t tyty = ty*ty;
    Fvec_t txtx1 = txtx + ONE;
    Fvec_t h = txtx + tyty;
    Fvec_t t = sqrt(txtx1 + tyty);
    Fvec_t h2 = h*h;
    Fvec_t qp0t = qp0*t;

    cnst c1=0.0136, c2=c1*0.038, c3=c2*0.5, c4=-c3/2.0, c5=c3/3.0, c6=-c3/4.0;

    Fvec_t s0 = (c1+c2*st.logRadThick + c3*h + h2*(c4 + c5*h +c6*h2)) *qp0t;
    Fvec_t a = (ONE+mass2*qp0*qp0t)*st.RadThick*s0*s0;

    CovV &C = track.C;
    C.C22 += txtx1*a;
    C.C32 += tx*ty*a; C.C33 += (ONE+tyty)*a;
}
Let's vectorize c=a+b

```c
float a[1000];
float b[1000];
float c[1000];

// add
for ( int i = 0; i < 1000; i++ ) {
    c[i] = a[i]+b[i];
}
```

```c
float aV[250];
float bV[250];
float cV[250];

// copy
for( int i = 0; i < 250; i++ ) {
    for( int iv = 0; iv < 4; iv++ ) {
        aV[i][iv] = a[i*4+iv];
        bV[i][iv] = b[i*4+iv];
    }
}
```

```c
// add
for( int i = 0; i < 250; i++ ) {
    cV[i] = aV[i]+bV[i];
}
```

```c
// copy back
for( int i = 0; i < 250; i++ ) {
    for( int iv = 0; iv < 4; iv++ )
        c[i*4+iv] = cV[i][iv];
}
```
Task:
- vector $r = \{x, y, z\}$
- calculate vector sum $r_c = r_a + r_b$
- for many pairs of vectors: $r_{a1}, r_{a2}, r_{a3}, \ldots; r_{b1}, r_{b2}, r_{b3}, \ldots$
Vectorization Approaches (2/3)

Task:
- vector $r = \{x, y, z\}$
- calculate vector sum $r_c = r_a + r_b$
- for many pairs of vectors: $r_{a1}$, $r_{a2}$, $r_{a3}$,...; $r_{b1}$, $r_{b2}$, $r_{b3}$,...

Inside the subtask (pair)

\[
\begin{align*}
  &x_{a1} & y_{a1} & z_{a1} & - \\
  + & x_{b1} & y_{b1} & z_{b1} & - \\
  = & x_{c1} & y_{c1} & z_{c1} & - \\
\end{align*}
\]

\[
\begin{align*}
  &x_{a2} & y_{a2} & z_{a2} & - \\
  + & x_{b2} & y_{b2} & z_{b2} & - \\
  = & x_{c2} & y_{c2} & z_{c2} & - \\
\end{align*}
\]

\[
\begin{align*}
  &x_{a3} & y_{a3} & z_{a3} & - \\
  + & x_{b3} & y_{b3} & z_{b3} & - \\
  = & x_{c3} & y_{c3} & z_{c3} & - \\
\end{align*}
\]

... ... ...

- Usually is not optimal
- Do not scale
Vectorization Approaches (3/3)

Task:
• vector \( r = \{x, y, z\} \)
• calculate vector sum \( r_c = r_a + r_b \)
• for many pairs of vectors: \( r_{a1}, r_{a2}, r_{a3},...; r_{b1}, r_{b2}, r_{b3},... \)

Between the subtasks

\[
\begin{align*}
X_{a1} & \quad X_{a2} & \quad X_{a3} & \quad X_{a4} \\
X_{b1} & \quad X_{b2} & \quad X_{b3} & \quad X_{b4} \\
\vdots & & & \\
X_{c1} & \quad X_{c2} & \quad X_{c3} & \quad X_{c4} \\
\end{align*}
\]

\[
\begin{align*}
y_{a1} & \quad y_{a2} & \quad y_{a3} & \quad y_{a4} \\
y_{b1} & \quad y_{b2} & \quad y_{b3} & \quad y_{b4} \\
\vdots & & & \\
y_{c1} & \quad y_{c2} & \quad y_{c3} & \quad y_{c4} \\
\end{align*}
\]

\[
\begin{align*}
Z_{a1} & \quad Z_{a2} & \quad Z_{a3} & \quad Z_{a4} \\
Z_{b1} & \quad Z_{b2} & \quad Z_{b3} & \quad Z_{b4} \\
\vdots & & & \\
Z_{c1} & \quad Z_{c2} & \quad Z_{c3} & \quad Z_{c4} \\
\end{align*}
\]

• Full vectorization if no branches
• Scales with big number of tasks
Array of Structures (AoS)

```c
struct TDataElement {
    float x, y, z;
};
TDataElement data[8];
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>

- Vectorization needs data movement
- Padding wastes the memory
+ x, y, z in the same cache line
+ Intuitively structured code

Structure of Arrays (SoA)

```c
struct TData {
    float x[8], y[8], z[8];
};
TData data;
```

```.x..y..y..y..z..z..z..z..```

+ The data is already grouped for vectorization
+ Compact data placement
- x, y, z in three different cache lines
- Confusing code

Array of Structures of Arrays

```c
struct TDataVecElement {
    float x[vecLen], y[vecLen], z[vecLen];
};
TDataVecElement data[2];
```

```.x..y..y..y..z..z..z..x..x..x..```

+ The data is already grouped for vectorization
+ Compact data placement
- x, y, z in three different cache lines
- Confusing code
Matrix Problem

Scalar

\[
\begin{bmatrix}
a_{00} & a_{01} & a_{02} \\
a_{10} & a_{11} & a_{12} \\
a_{20} & a_{21} & a_{22}
\end{bmatrix}
\]

\[
\begin{bmatrix}
\sqrt{a_{00}} & \sqrt{a_{01}} & \sqrt{a_{02}} \\
\sqrt{a_{10}} & \sqrt{a_{11}} & \sqrt{a_{12}} \\
\sqrt{a_{20}} & \sqrt{a_{21}} & \sqrt{a_{22}}
\end{bmatrix}
\]

\[
= ?
\]

SIMD

\[
\begin{array}{l}
\text{for( int } i = 0; i < N; i++ ) \{ \\
\quad \text{for( int } j = 0; j < N; j++ ) \{ \\
\quad \quad c[i][j] = sqrt(a[i][j]); \\
\quad \} \\
\} \\
\]

Task:
- Write the SIMD part of the code using the SIMD-header file.
- Compare the processing time and the results.
- Can you avoid copying?
**Exercise 2.4.3**

**Quadratic Equations Problem**

\[ ax^2 + bx + c = 0 \]

\[ x_{\text{max}} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]

**Scalar**

```c
for (int i = 0; i < N; i++) {
    float det = b[i]*b[i] - 4*a[i]*c[i];
    x[i] = (-b[i]+sqrt(det))/(2*a[i]);
}
```

**SIMD**

```c
for(int i=0; i<N; i++) {
    float det = b[i]*b[i] - 4*a[i]*c[i];
    x[i] = (-b[i]+sqrt(det))/(2*a[i]);
}
```

**Task:**

- Write the SIMD code for the root calculation using SIMD intrinsics and copying the data to SIMD vectors.
- Write the SIMD code using SIMD intrinsics and casting the data from the scalar arrays to SIMD vectors (use reinterpret_cast for this). Compare the time with a previous task.
- Write the SIMD code using header files and copying the data to SIMD vectors. Compare the time with previous tasks.
- Write the SIMD code using header files and casting the data from the scalar arrays to SIMD vectors (use reinterpret_cast for this). Compare the time with previous tasks.
- Put NVectors = 10000000; and NIterOut=10. Compare times and speed up factors with the previous results.
template< typename T >
T Sum(const T* data, const int N)
{
    T sum = 0;
    const T* end = data + N;
    for ( int i = 0; i < N; ++i )
        sum = sum ^ data[i];
    return sum;
}

Task:
- How to vectorize?
- What is max speed up?
- Is it possible to vectorize without SIMD-intrinsics?