SIMD Exploitation in (JIT) Compilers

Hiroshi Inoue, IBM Research - Tokyo
What’s SIMD?

- **Single Instruction “Multiple Data”**
- **Same operations applied for multiple elements in a vector register**

**scalar instruction**

```
input 1
input 2
output
```

```
add gr1, gr2, gr3
```

**SIMD instruction**

```
input 1
input 2
output
```

```
vadd vr1, vr2, vr3
```

**vector register**

**SIMD instructions**
SIMD is all around us

- **Intel x86**
  - MMX (since MMX Pentium released in 1997)
  - SSE, SSE2, SSE3, SSE4
  - AVX, AVX2
  - Xeon Phi (co-processor)

- **PowerPC**
  - VMX (a.k.a. AltiVec, Velocity Engine), VSX
  - SPU of Cell BE (SIMD only ISA)

- **ARM**
  - NEON, NEON2
What’s SIMD?

- Single Instruction “Multiple Data”
- Same operations applied for multiple elements in a vector register

**Scalar Instruction**

```
add gr1, gr2, gr3
```

**SIMD Instruction**

```
vadd vr1, vr2, vr3
```
SIMD is all around us

- **Intel x86**
  - MMX (since MMX Pentium released in 1997)
  - SSE, SSE2, SSE3, SSE4
  - AVX, AVX2
  - Xeon Phi (co-processor)

- **PowerPC**
  - VMX (a.k.a. Altivec, Velocity Engine)
  - SPU of Cell BE (SIMD only ISA)

- **ARM**
  - NEON, NEON2
So what?

- SIMD yields higher peak performance
  - e.g. Peak flops become 4x for double fp with 256-bit vector registers (AVX)

So, can we get 4x speed up by just buying SIMD processors?

Then, can we get 4x speed up by recompiling my source code?
Outline

- Background - What’s SIMD?

- Approaches for SIMD programming
  - Explicit SIMD parallelization
  - Automatic loop vectorization
  - Programming models suitable for SIMD

- Summary
Manual programming with SIMD

- We can write SIMD instructions explicitly using inline assembly!
- Compiler support can make the explicit SIMD programming (somewhat) easier
  - data types to represent vector register
  - built-in methods that are directly mapped onto SIMD instructions (called “intrinsics”)
- Most of processor vendors define C/C++ language extensions for their SIMD ISA
Examples of programming with SIMD intrinsics

Original scalar loop

```c
for (i = 0; i < 1024; i++) c[i] = a[i] + b[i];
```

Vectorized with AVX2 intrinsics

```c
for (i = 0; i < 1024; i+=8) {
    __m256i v1 = _mm256_loadu_si256((__m256i *)(a+i));
    __m256i v2 = _mm256_loadu_si256((__m256i *)(b+i));
    __m256i v3 = _mm256_add_epi32(v1, v2);
    v3 = _mm256_loadu_si256((__m256i *)(c+i), v3);
}
```

**vector int** represents a vector register, which contains 4 integers

Vectorized with VMX intrinsics

```c
for (i = 0; i < 1024; i+=4) {
    vector int v1 = vec_ld(0, a+i);
    vector int v2 = vec_ld(0, b+i);
    vector int v3 = vec_add(v1, v2);
    vec_st(v3, 0, c+i);
}
```

32-bit integer arrays

8 (= 256 / 32) elements at once

4 (= 128 / 32) elements at once

vec_add() is mapped onto a SIMD add instruction
Explicit SIMD Programming – Pros & Cons

Explicit SIMD Programming with intrinsics:

😊 can achieve **best possible performance** in most cases

😊 is easier than assembly programming (register allocation, instruction scheduling etc.)

😢 is still very **hard to program, debug and maintain**

😢 depends on underlying processor architecture and is **not portable** among different processors

😢 is not suitable for platform-neutral languages such as Java and scripting languages

Often, it also requires change in algorithms and data layout for efficient SIMD exploitation
Outline

- Background - What’s SIMD?
- Approaches for SIMD programming
  - Explicit SIMD parallelization
  - Automatic loop vectorization
  - Programming models suitable for SIMD
- Summary
Can compilers help more?

- Yes, **automatic loop vectorization** has been studied for vector computers and later for SIMD instructions.

- Compiler analyzes scalar loop and translates it to vector processing if possible.
  - Major reasons that prevent vectorization include:
    - loop-carried dependency
    - control flow (e.g. if statement) in loop body
    - memory alignment issue
    - method call in loop body
Example of automatic loop vectorization

Original scalar loop

```c
for (i = 0; i < N; i++) c[i] = a[i] + b[i];
```

Compiler needs to analyze loops:

- input and output vectors may overlap each other
  - loop-carried dependency
- vectors may not properly aligned
- loop count \((N)\) may not be a multiple of the vector size

Compiler generates guard code to handle each case or just gives up vectorization
Automatic loop vectorization – Pros & Cons

Automatic loop vectorization:

😊 does not require source code changes

😢 performance gain is limited compared to hand vectorization

Average performance gain of many loops over scalar (non-SIMD) by automatic and manual vectorization [1]

<table>
<thead>
<tr>
<th>Method</th>
<th>XLC</th>
<th>ICC</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Vectorization</td>
<td>1.66</td>
<td>1.84</td>
<td>1.58</td>
</tr>
<tr>
<td>Transformations</td>
<td>2.97</td>
<td>2.38</td>
<td></td>
</tr>
<tr>
<td>Intrinsics</td>
<td>3.15</td>
<td>2.45</td>
<td></td>
</tr>
</tbody>
</table>

Higher productivity but lower expected performance gain compared to the explicit approach

Programmer can help automatic loop vectorization

- By changing algorithms and data layout
  - algorithm
    - e.g. SOR → red-black SOR
  - data layout
    - e.g. array of structures → structure of arrays

- By adding explicit declarations or pragmas
  - “restrict” keyword of C99
    - declaration to show there is no aliasing
      ```c
      void func(double *restrict a, double *restrict b, double *restrict c)
      { /* a[], b[], and c[] are independent in this function */ }
      ```
  - standard/non-standard pragmas
    - e.g. #pragma omp simd (OpenMP 4.0), #pragma simd (icc), #pragma disjoint (xlc) etc
AOS and SOA

- Key data layout transformation for efficient SIMD execution by avoiding discontiguous (gather/scatter) memory accesses

Array of Structures (AOS)

x0 y0 z0 x1 y1 z1 x2 y2 z2 x3 y3 z3 x4 y4 z4 x5 y5 ...

Structure of arrays (SOA)

x0 x1 x2 x3 x4 ... y0 y1 y2 y3 y4 ... z0 z1 z2 z3 z4 ...

Hybrid (often performs best for hand vectorization)

x0 x1 x2 x3 y0 y1 y2 y3 z0 z1 z2 z3 x4 x5 x6 x7 y4 ...

(assuming 4-way SIMD)
Automatic vectorization for JIT compilers

- Automatic vectorization can be implemented in JIT compilers as well as static compilers

**Pros & Cons**

😊 JIT compiler can select best SIMD ISA at runtime (e.g. SSE or AVX)

رياضات The analysis for loop vectorization is often too costly for JIT compilation

- To avoid excessive compilation time in a JIT compiler, it is possible to execute analysis offline and embedded information in bytecode [2]

Tradeoff between programmability and performance

Is there a good way to *balance* performance and programmability?

Explicit SIMD parallelization

- higher performance
- lower programmability
- lower portability

Automatic loop vectorization

- lower performance
- higher programmability
- higher portability
Outline

- Background - What’s SIMD?
- Approaches for SIMD programming
  - Explicit SIMD parallelization
  - Automatic loop vectorization
  - Programming models suitable for SIMD
- Summary
New programming models suitable for SIMD programming

- **Stream processing**
  - small programs (*kernel functions*) are applied for each element in data streams (e.g. StreamIt, Brook, RapidMind)

- **SIMT (Single Instruction Multiple Threads)**
  - execution model for GPU introduced by Nvidia
  - each slot of SIMD instruction works as an independent thread (e.g. CUDA, OpenCL)

→ Programmer and Language runtime system collaborate to vectorize the code
  - Programmer identifies and explicitly shows parallelism
  - Language runtime is responsible for optimizing the code for the underlying architecture
Example of OpenCL kernel [3]

### Traditional loops
```c
void trad_mul(int n,  
    const float *a,  
    const float *b,  
    float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];  
}
```

### Data Parallel OpenCL
```c
kernel void dp_mul(  
    global const float *a,  
    global const float *b,  
    global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over “n” work-items
```

- Programmer write a program for each data element as scalar code (called “kernel”)
- Runtime applies the kernel for all data elements

[3] Neil Trevett, OpenCL BOF at SIGGRAPH 2010,  
SIMT architecture, which extends SIMD

- Today’s GPU architecture extends SIMD for efficient execution of SIMT code
- Each thread executing kernel is mapped onto a slot of SIMD instructions
  - gather/scatter memory access support
  - predicated execution support to convert control flow to data flow

SIMD ISA of general-purpose processors are also going toward a similar direction (e.g. AVX2 supports gather and scatter)
Other programming model for SIMD

- Framework or language for explicit SIMD parallelization with abstracted SIMD for better portability
  - Boost.SIMD [4], VecImp [5]

- SIMT processing model designed for SIMD instructions of general-purpose processors
  - Intel SPMD Program Compiler (ispc) [6] exploits SIMT (they call SPMD) model for programming SSE and AVX

Summary

- Categorized SIMD programming techniques into three approaches
  - Explicit SIMD parallelization
  - Automatic loop vectorization
  - Programming models suitable for SIMD

- No one-fit-all solution; each approach has benefits and drawbacks