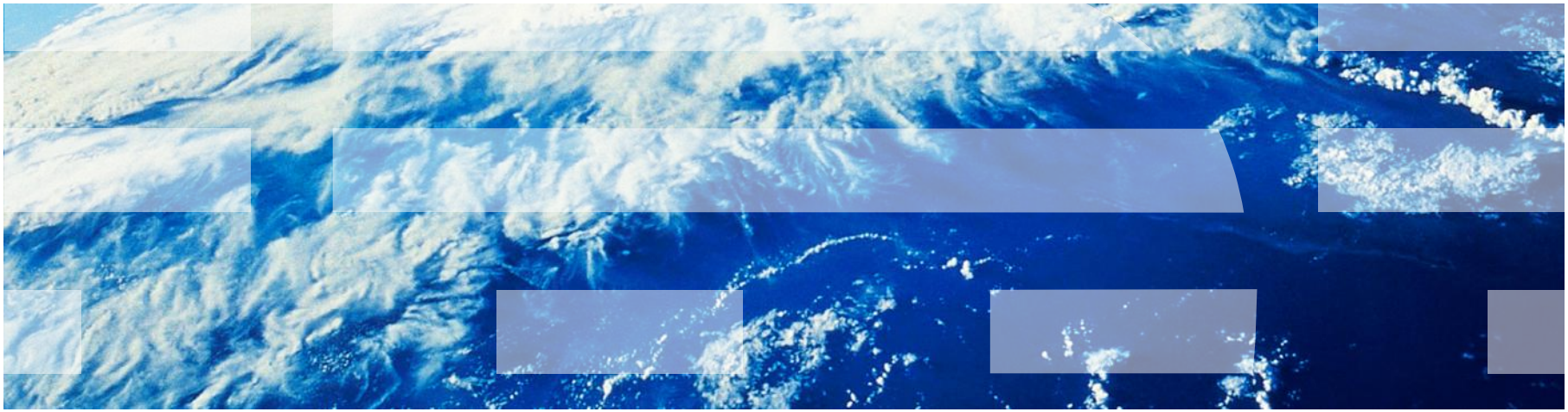
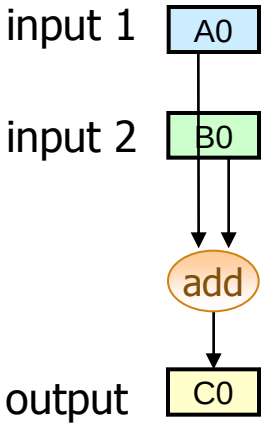
SIMD Exploitation in (JIT) Compilers

Hiroshi Inoue, IBM Research - Tokyo



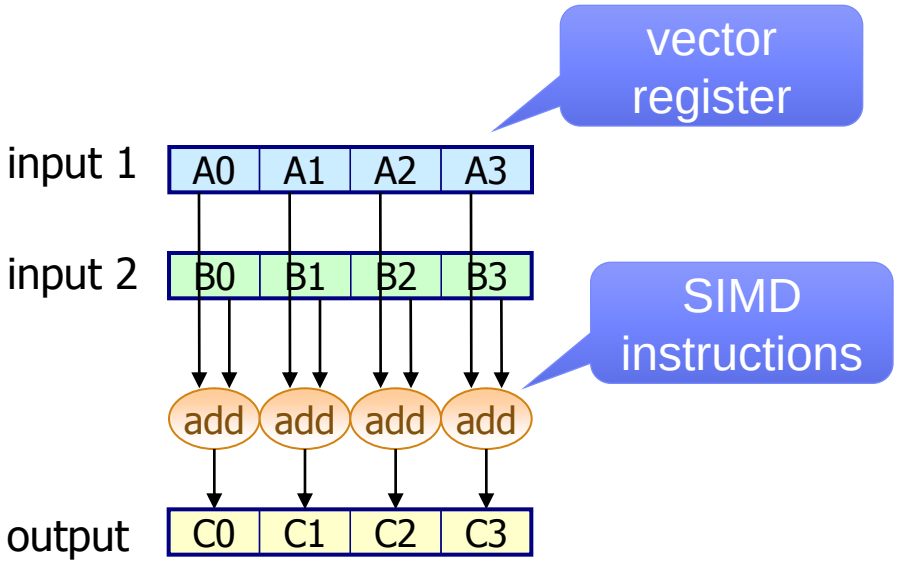
What's SIMD?

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




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

scalar instruction




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

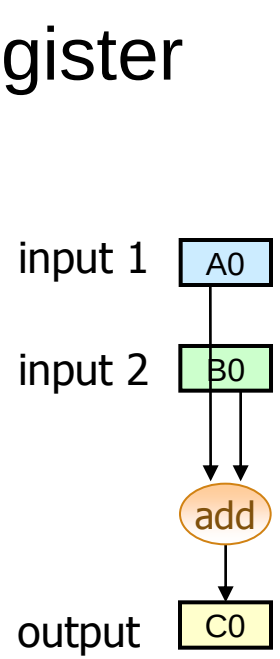
SIMD instruction

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

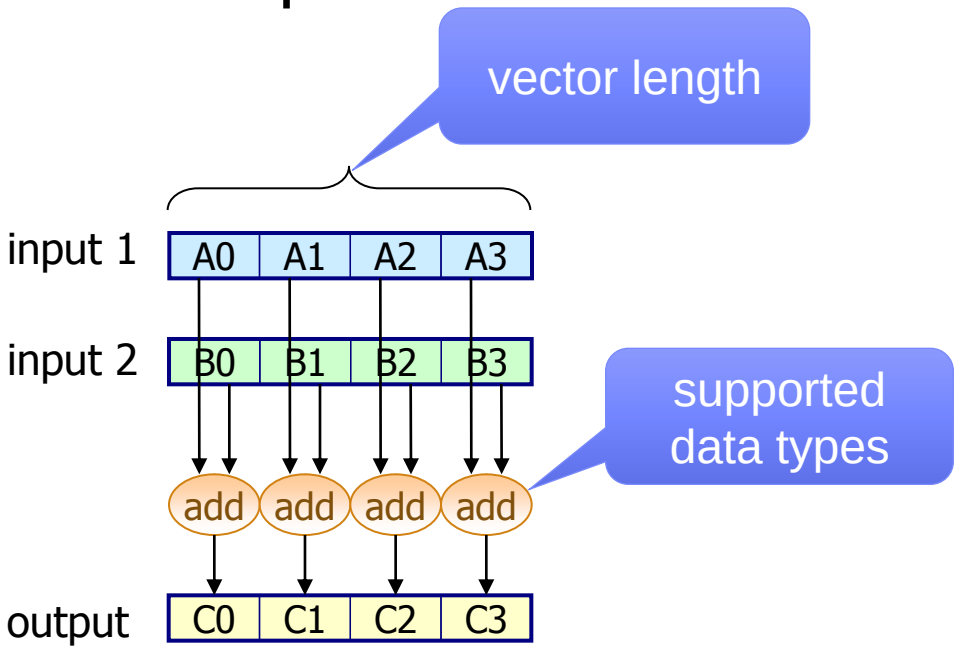
What's SIMD?

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



`add gr1,gr2,gr3`

scalar instruction



`vadd vr1,vr2,vr3`

SIMD instruction

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. Alti
  - SPU of Cell BE (SIMD only ISA)
- ARM
  - NEON, NEON2

64-bit vector registers (shared with FPRs)
integer types only

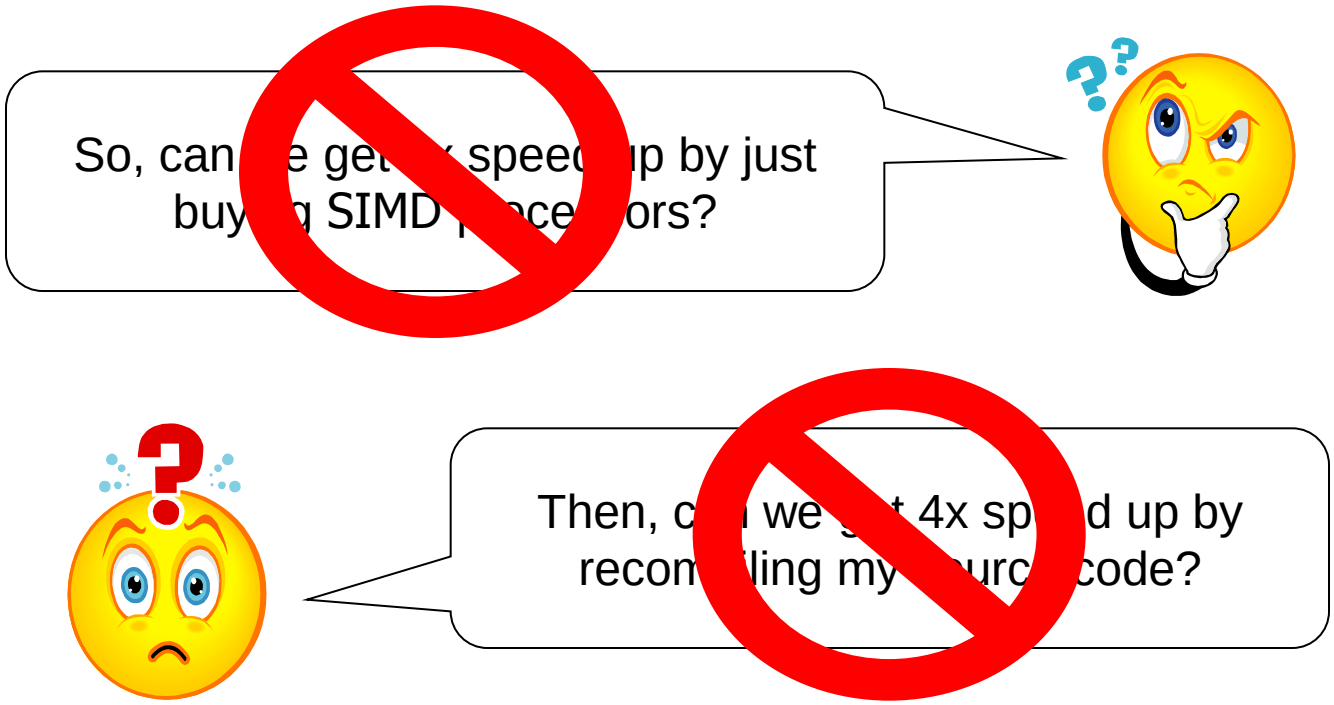
128-bit vector registers
integer, single fp, double fp

256-bit vector registers
single fp, double fp, integer

512-bit vector registers
single fp, double fp

So what?

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



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
  - –build-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

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

32-bit integer arrays

Vectorized with AVX2 intrinsics

```
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);
    _mm256_storeu_si256((__m256i *)(c+i), v3);  }
```

8 (= 256 / 32) elements at once

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

Vectorized with VMX intrinsics

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

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

```
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]

Method	XLC	ICC	GCC
Auto Vectorization	1.66	1.84	1.58
Transformations	2.97	2.38	
Intrinsics	3.15	2.45	

on POWER7 on Nehalem

➔higher productivity but lower expected performance gain compared to the explicit approach

[1] Maleki *et al.* "An Evaluation of Vectorizing Compilers", PACT 2011

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

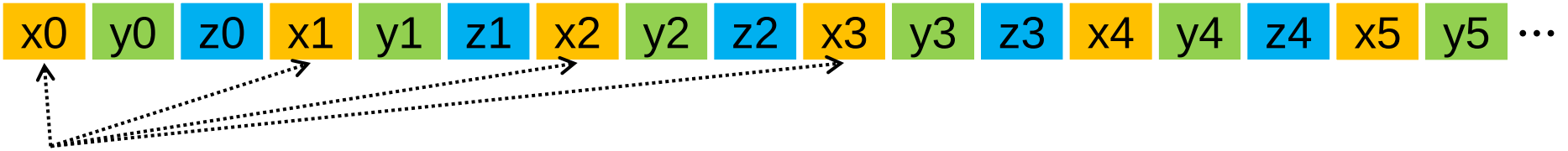
```
void func(double *restrict a, double *restrict b, double *restrict c)
{  /* a[], b[], and c[] are independent in this function  */ }
```

  - –standard/non-standard pragmas
    - e.g. #pragma omp simd (OpenMP 4.0), #pragma simd (icc), #pragma disjoint (xlc) etc

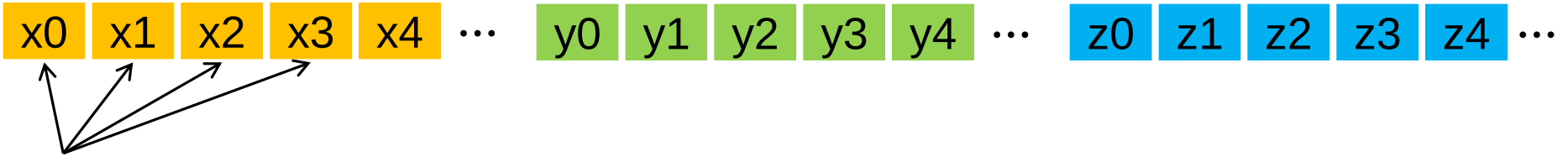
AOS and SOA

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

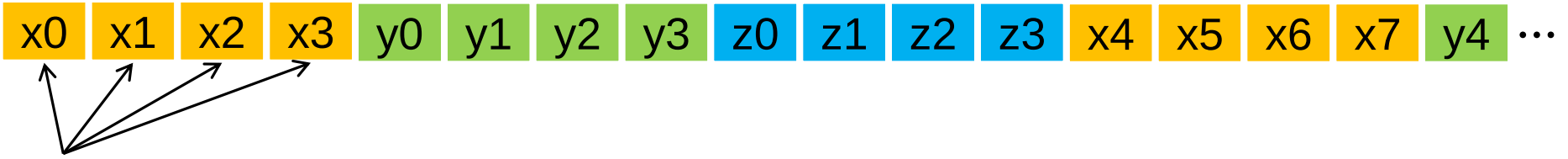
Array of Structures (AOS)



Structure of arrays (SOA)



Hybrid (often performs best for hand vectorization)



(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)

☹ 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]

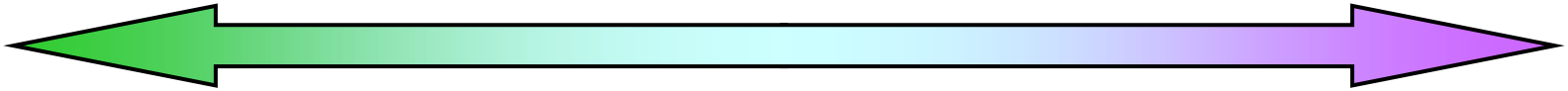
[2] Nuzman *et al.* "Vapor SIMD: Auto-vectorize once, run everywhere", CGO 2011

Tradeoff between programmability and performance

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

Explicit SIMD parallelization

Automatic loop vectorization



☺ higher performance
☹ lower programmability
☹ lower portability

☹ 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

```
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

```
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, https://www.khronos.org/assets/uploads/developers/library/2010_siggraph_bof_opencl/OpenCL-BOF-Intro-and-Overview_SIGGRAPH-Jul10.pdf

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

[4] Estérie *et al.* "Boost.SIMD: generic programming for portable SIMDization", PACT 2012
[5] Leißa *et al.* "Extending a C-like language for portable SIMD programming", PPoPP 2012
[6] Pharr *et al*. "ispc: A SPMD Compiler for High-Performance CPU Programming", Innovative Parallel Computing, 2012.

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