

Vectorization and the JVM

Vladimir Ivanov
HotSpot JVM Compiler
Oracle Corp.
April 8, 2017

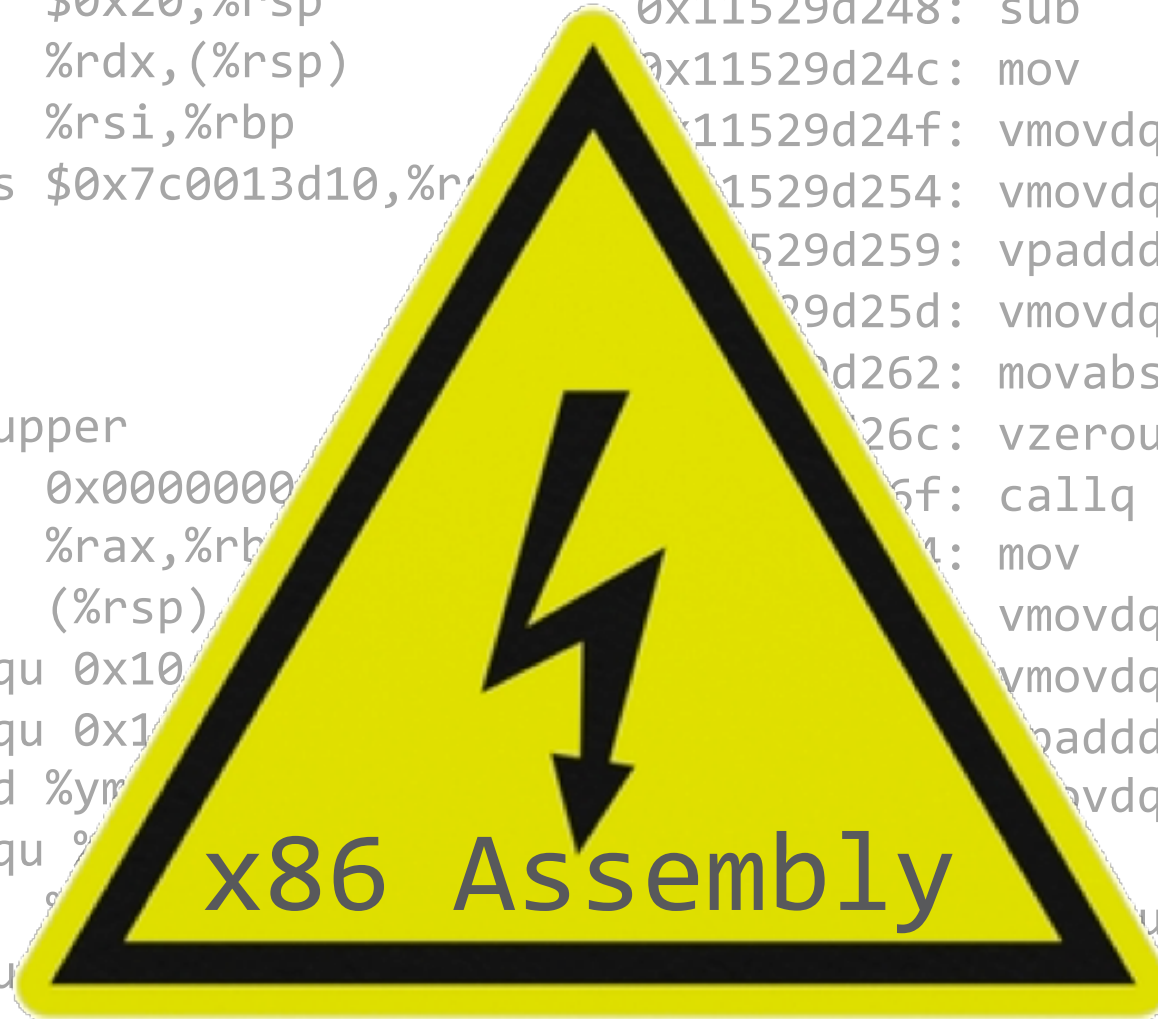
Java
Your
Next
(Cloud)



Agenda

- SIMD ISA extensions
 - packed vectors on x86
- JVM
 - auto-vectorization, intrinsics
- Future
 - JDK 9
 - Vector API

0x11529c8c0: mov %eax, -0x16000(%rsp)	0x11529d240: mov %eax, -0x16000(%rsp)
0x11529c8c7: push %rbp	0x11529d247: push %rbp
0x11529c8c8: sub \$0x20,%rsp	0x11529d248: sub \$0x30,%rsp
0x11529c8cc: mov %rdx, (%rsp)	0x11529d24c: mov %rcx,%rbp
0x11529c8d0: mov %rsi,%rbp	0x11529d24f: vmovdqu 0x10(%rsi),%ymm0
0x11529c8d3: movabs \$0x7c0013d10,%rsi	0x11529d254: vmovdqu 0x10(%rdx),%ymm1
0x11529c8dd: nop	0x11529d259: vpaddd %ymm0,%ymm1,%ymm0
0x11529c8de: nop	0x11529d25d: vmovdqu %ymm0, (%rsp)
0x11529c8df: nop	0x11529d262: movabs \$0x7c0013d10,%rsi
0x11529c8e0: vzeroupper	0x11529d26c: vzeroupper
0x11529c8e3: callq 0x00000000	0x11529d26f: callq 0x000000001152418a0
0x11529c8e8: mov %rax,%rbx	0x11529d274: mov %rax,%rbx
0x11529c8eb: mov (%rsp)	0x11529d278: vmovdqu 0x10(%rbp),%ymm1
0x11529c8ef: vmovdqu 0x10(%rsi),%ymm0	0x11529d27c: vmovdqu (%rsp),%ymm0
0x11529c8f5: vmovdqu 0x10(%rdi),%ymm0	0x11529d280: vpaddd %ymm0,%ymm1,%ymm0
0x11529c8fa: vpaddd %ymm0,%ymm1,%ymm0	0x11529d284: vmovdqu %ymm0,0x10(%rbx)
0x11529c8fe: vmovdqu %ymm0,%ymm1	0x11529d288: mov %rbx,%rax
0x11529c903: mov %rax,%rbx	0x11529d28c: vzeroupper
0x11529c906: vzeroupper	0x11529d290: sub \$0x30,%rsp
0x11529c909: add \$0x20,%rsp	0x11529d294: pop %rbp
0x11529c90d: pop %rbp	0x11529d295: test %eax, -0xb78729b(%rip)



Assembly Syntax

AT&T

```
mov 0x10(%src),%dst
```

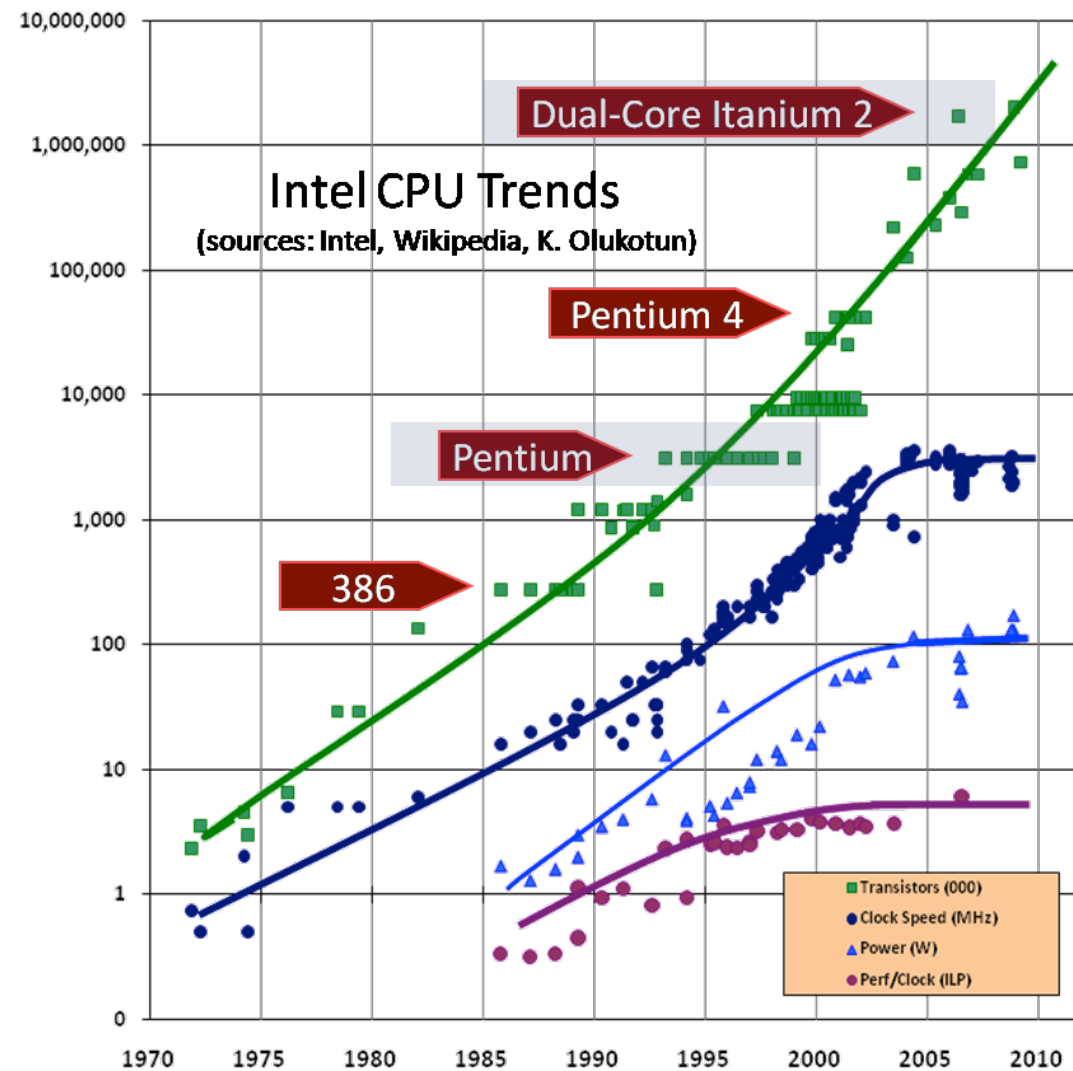


VS

```
mov dst,[src+10h]
```

Intel

“The Free Lunch Is Over”, Herb Sutter, 2005



Going Parallel

- Machines < 10^3 - 10^6 (servers)
 - Hadoop (Map/Reduce), Apache Spark
- Cores/hardware threads < 10s-100s (threads)
 - Java Stream API
 - Fork/Join framework
- **CPU SIMD extensions** < 10s (elements)
 - x86: SSE ..., AVX, ..., AVX-512

Going Parallel: CPUs vs Co-processors

- CPUs
 - SIMD ISA extensions (**S**ingle **I**nstruction-**M**ultiple **D**ata)
 - threads (**M**ultiple **I**nstructions-**M**ultiple **D**ata)
- Co-processors
 - GPUs
 - FPGAs
 - ASICs
 - Data Analytics Accelerator (DAX) on SPARC

SIMD vs MIMD

- Machines
— up to 12 cards / server
 $< 10^3$ - 10^6 12x (servers)
- **Intel Xeon Phi**
— 4 threads x 72 cores
 < 10 s-100s 288 (threads)
- AVX-512
— 2 units / core
 < 10 s 16 SP (elements)

SIMD vs MIMD

- Machines
 - up to 12 cards / server
 - **Intel Xeon Phi**
 - 4 threads x 72 cores
 - AVX-512
 - 2 units / core
- | | | |
|-------------------|-------|------------|
| $< 10^3$ - 10^6 | 12x | (servers) |
| < 10 s-100s | 288 | (threads) |
| | vs | |
| < 10 s | 16 SP | (elements) |

SIMD vs MIMD

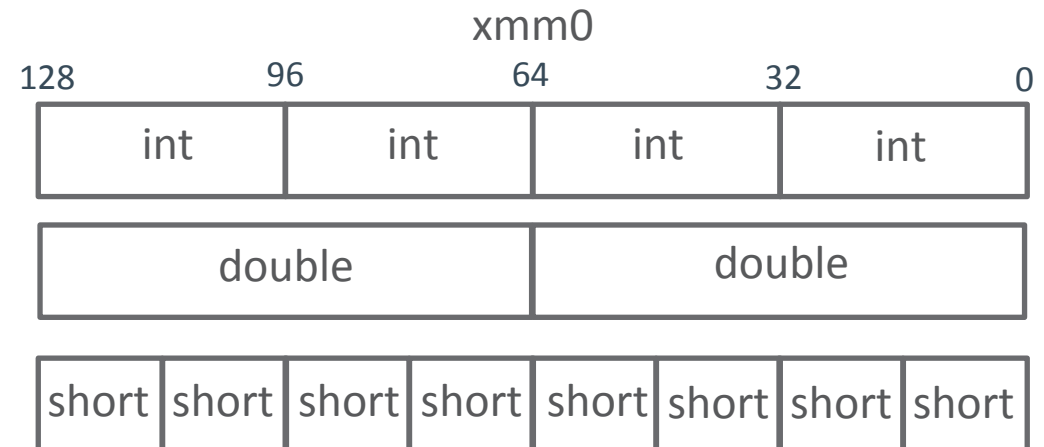
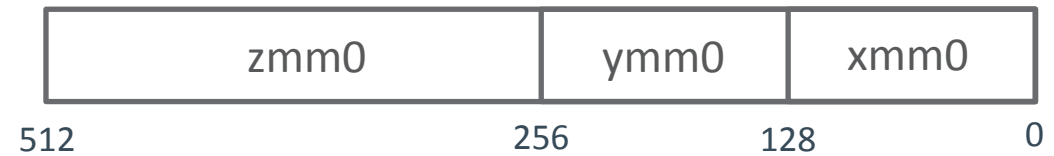
• Machines — up to 12 cards / server	$< 10^3$ - 10^6	12x	(servers)
• Intel Xeon Phi — 4 threads x 72 cores	< 10 s-100s	288 vs 16 SP	(threads) (elements)
• AVX-512 — 2 units / core	< 10 s		
			4608-way

SIMD today

- x86: MMX, SSE, AVX
 - 8 64-bit registers (MMX) to 32 512-bit registers (AVX-512)
- ARM: NEON
 - 32 128-bit registers
- SPARC: VIS
 - 32 64-bit registers
- POWER: VMX/Altivec
 - 32 128-bit registers

x86 SIMD Extensions

- Wide (multi-word) registers
 - 128-bit (xmm)
 - 256-bit (ymm)
 - 512-bit (zmm)
- Instructions on packed vectors
 - packed in a register or memory location
 - 2 x double, 4 x int, 8 x short
 - hard-coded vector size



x86 SIMD Extensions

```
// Load A[i:i+3]
```

```
vmovdqu 0x10(%rcx,%rdx,4),%xmm0
```

```
// Load B[i:i+3]
```

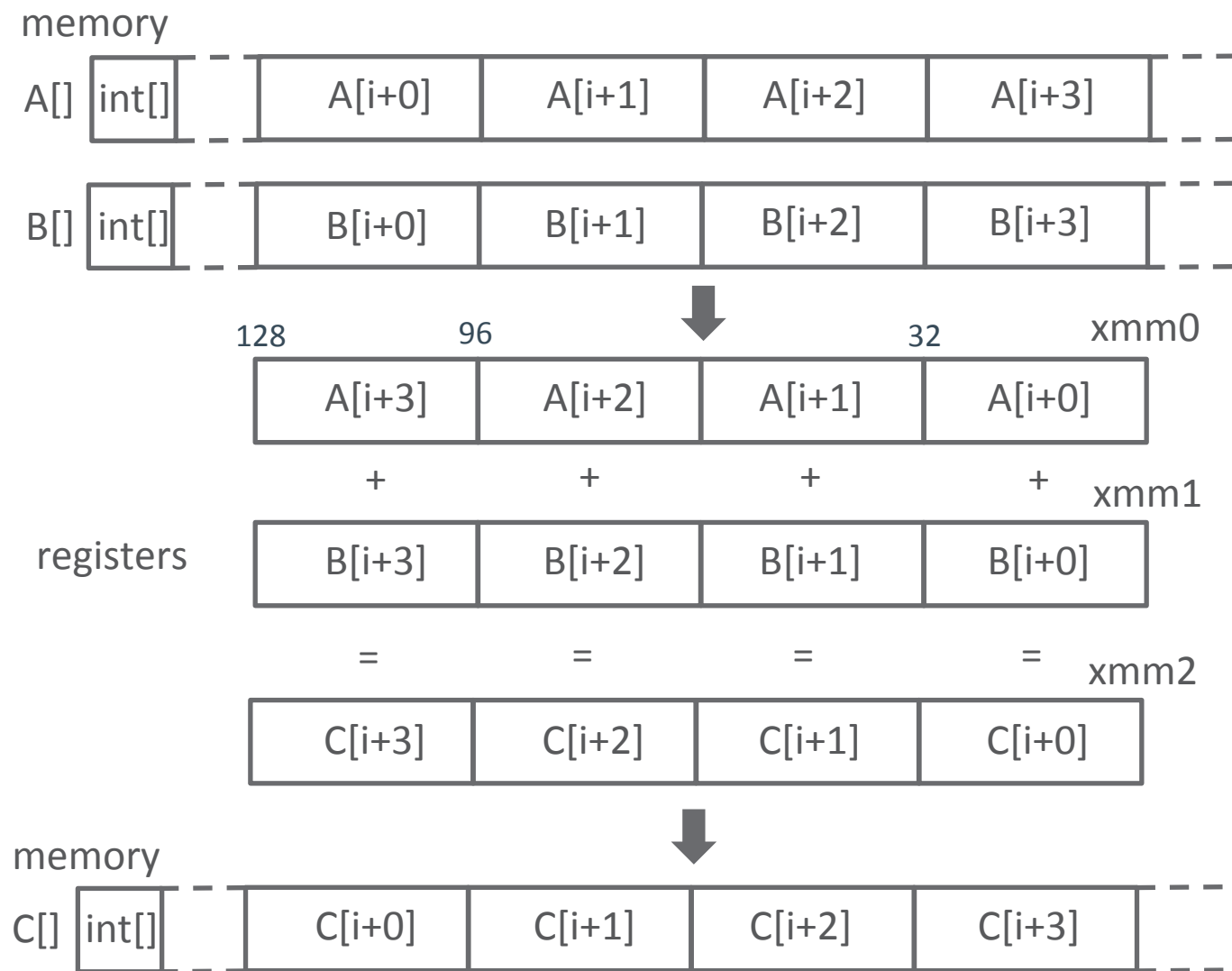
```
vmovdqu 0x10(%r10,%rdx,4),%xmm1
```

```
// A[i:i+3] + B[i:i+3]
```

```
vpaddq %xmm0,%xmm1,%xmm2
```

```
// Store into C[i:i+3]
```

```
vmovdqu %xmm2,0x10(%r8,%rdx,4)
```



Year	Name	Reg Size (bit)	
1997	MMX	64	mm0-7
1999	SSE	128	xmm0-7
2001	SSE2	128	xmm0-15
2004	SSE3	128	xmm0-15
2006	SSSE 3	128	xmm0-15
2006	SSE 4.1	128	xmm0-15
2008	SSE 4.2	128	xmm0-15
2011	AVX	256	ymm0-15
2013	AVX2	256	ymm0-15
2013	FMA3	256	ymm0-15
2015	AVX-512	512	zmm0-31 (k0-7)

How to utilize SIMD instructions?

Vectorization techniques

- Automatic
 - sequential languages and practices gets in the way
- Semi-automatic
 - Give your compiler/runtime hints and hope it vectorizes
 - OpenMP 4.0 `#pragma omp simd`
- Code explicitly
 - SIMD instruction intrinsics

Problem

If the code is compiled for a **particular instruction set** then it will be **compatible** with all CPUs that **support** this instruction set or any higher instruction set, but **possibly not** with **earlier** CPUs.

SSE 4.2 << AVX-512

CPU Dispatching

Idea:

Make critical parts of the code in **multiple versions** for different CPUs.

- For example, provide:
 - AVX2 & SSE 4.2 specializations
 - generic version that is compatible with old microprocessors
- The program should automatically detect which instruction set is supported and choose the appropriate version.

CPU Dispatching

“It is quite **expensive** - in terms of development, testing and maintenance - to make a piece of code in multiple versions, each carefully optimized and fine-tuned for a particular set of CPUs. “

“Optimizing software in C++”, Agner Fog

CPU dispatching: Common pitfalls

- Optimizing for present processors rather than future processors
- Thinking in terms of specific processor models rather than processor features
- Assuming that processor model numbers form a logical sequence
- Failure to handle unknown processors properly
- Underestimating the cost of keeping a CPU dispatcher updated
- Making too many branches
- Ignoring virtualization

[“Optimizing software in C++”, Agner Fog](#)

JVM and SIMD today

JVM is in a good position:

1. Java bytecode is platform-agnostic
2. CPU probing at runtime (at startup)
 - knows everything about the hardware it executes at the moment
3. Dynamic code generation
 - only use instructions which are available on the host

JVM and SIMD today

- Hotspot supports some of x86 SIMD instructions
- Automatic vectorization of Java code
 - Superword optimizations in HotSpot C2 compiler to derive SIMD code from sequential code
- JVM intrinsics
 - Array copying, filling, and comparison

JVM Intrinsic

JVM Intrinsic

“A method is intrinsicified if the HotSpot VM replaces the **annotated method with hand-written assembly and/or hand-written compiler IR** -- a compiler intrinsic -- to improve performance.”

[@HotSpotIntrinsicCandidate JavaDoc](#)

```
public final class java.lang.Class<T> implements ... {  
    @HotSpotIntrinsicCandidate  
    public native boolean isInstance(Object obj);  
}
```

Vectorized JVM Intrinsics

- Array copy
 - `System.arraycopy()`, `Arrays.copyOf()`
- Array mismatch (@since 9)
 - `Arrays.mismatch`
 - `ArraysSupport.vectorizedMismatch()`

Auto-vectorization

by JIT-compiler

Exploiting Superword Level Parallelism with Multimedia Instruction Sets

Samuel Larsen and Saman Amarasinghe
MIT Laboratory for Computer Science
Cambridge, MA 02139
{slarsen,saman}@lcs.mit.edu

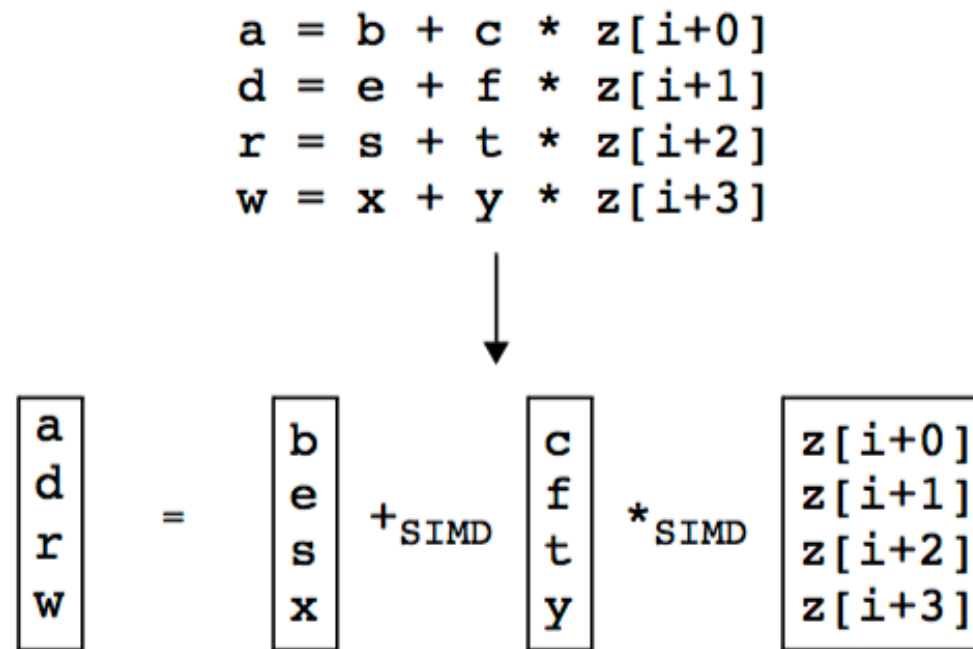


Figure 1: Isomorphic statements that can be packed and executed in parallel.

Vectorization: Prerequisites

SuperWord optimization is:

1. implemented only in C2 JIT-compiler

```
hotspot/src/share/vm/opto/c2_globals.hpp:
```

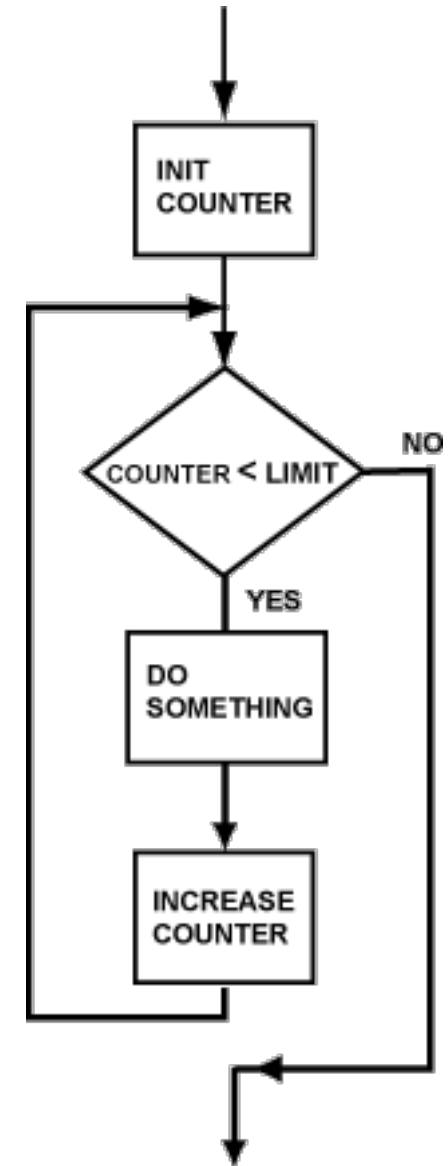
```
product(bool, UseSuperWord, true,  
        "Transform scalar operations into superword operations")
```

2. applied only to unrolled loops
 - unrolling is performed **only** for counted loops

Trip-Counted Loop

“Counted loops are all trip-counted loops, with exactly 1 trip-counter exit path (and maybe some other exit paths). The trip-counter exit is always last in the loop. The trip-counter have to stride by a constant; the exit value is also loop invariant.”

hotspot/src/share/vm/opto/loopnode.hpp:136



Trip-Counted Loop

- limit is loop invariant
- stride is constant (compile-time)

```
for (int i = start; i < limit; i+=stride) {  
    // Loop body  
}
```

```
int i = start;  
while (i < limit) {  
    // Loop body  
    i+=stride;  
}
```

How to detect?

1. `$ java ... -XX:+PrintCompilation -XX:+TraceLoopOpts ...`

–available only in debug builds

```
129  1  b    TripCountedLoop::test1 (28 bytes)
```

```
Counted Loop: N100/N83  limit_check predicated counted [0,100),+1 (-1 iters)
```

2. `$ java ... -XX:+PrintAssembly ...`

–and eyeball generated code

How to detect?

1. `$ java ... -XX:+PrintCompilation -XX:+TraceLoopOpts ...`

–available only in debug builds



```
129  1  b    TripCountedLoop::test1 (28 bytes)
```

```
Counted Loop: N100/N83  limit_check predicated counted [0,100),+1 (-1 iters)
```

2. `$ java ... -XX:+PrintAssembly ...`

–and eyeball generated code

Trip-Counted Loop?

```
for (int i = 0; i < 100; i++) { /*Loop body*/ }
```



```
$ java ... -XX:+TraceLoopOpts ...
```

Counted Loop: N100/N83 ... counted [0,100),+1 (-1 iters)

Trip-Counted Loop?

```
for (int i = start; i < 100; i++) { /*Loop body*/ }
```



Counted Loop: N104/N84 ... counted [**int**,**100**),+1 (-1 iters)

Trip-Counted Loop?

```
for (int i = start; i < end; i++) { /*Loop body*/ }
```



Counted Loop: N104/N84 ... counted [**int,int**),+1 (-1 iters)

Trip-Counted Loop?

```
for (int i = start;  
    i < end1 && i < end2;  
    i++) { ... }
```



Counted Loop: N104/N84 ... counted [**int,int**),+1 (-1 iters)

Trip-Counted Loop?

```
for (int i = start;  
    i < end1 || i < end2;  
    i++) { ... }
```



Loop: N101/N93 limit_check predicated sfpts={ 93 }

PartialPeel Loop: N101/N93 limit_check predicated sfpts={ 93 }

Counted Loop: N136/N64 counted [int,int),+1 (-1 iters)



Trip-Counted Loop?

```
for (int i = start; i < end; i+=2) { /*Loop body*/ }
```



Counted Loop: N108/N85 ... counted [**int,int**),+2 (-1 iters)

Trip-Counted Loop?

for (**int** i = start; i < end; **i+=d**) { */*Loop body*/* } ❌

for (**int** i = start; i < end; **i*=2**) { */*Loop body*/* } ❌

for (**int** i = start; i < end; i++) {
 ... **if** (...) { **end++**; } ...
}

Trip-Counted Loop?

for (**long** i = 0; i < 100; i++) {...} 

Trip-Counted Loop?

for (**long** l = 0; l < 100; l++) {...} ❌

for (**int** i = 0; i < 100; i++) {...} ✅

for (**byte** b = 0; b < 100; b++) {...} ❌

for (**short** s = 0; s < 100; s++) {...} ✅

for (**char** c = 0; c < 100; c++) {...} ✅

Trip-Counted Loop?

```
for (int i = 0; i < 100; i++) {  
    ... f(); // not inlined  
}
```



Counted Loop: ... counted $[0, 100)$, +1 (-1 iters) has_call has_sfpt

Trip-Counted Loop?

```
for (int i = 0; i < 100; i++) {  
    ... f(); // not inlined  
}
```



Counted Loop: ... counted [0,100),+1 (-1 iters) **has_call** **has_sfpt**

But no unrolling happens, hence no vectorization.



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

Loop unrolling (4 times)

```
for (int i = 0; i < MAX-4; i+=4) { // main loop
    A[i+0] = B[i+0] + C[i+0];
    A[i+1] = B[i+1] + C[i+1];
    A[i+2] = B[i+2] + C[i+2];
    A[i+3] = B[i+3] + C[i+3];
}
// post-loop
```

Loop unrolling

```
for (int i = 0; i < MAX-4; i+=4) {  
    A[i+0] = B[i+0] + C[i+0];  
    A[i+1] = B[i+1] + C[i+1];  
    A[i+2] = B[i+2] + C[i+2];  
    A[i+3] = B[i+3] + C[i+3];  
}
```

isomorphic

A[i:i+3] B[i:i+3] C[i:i+3]

Main loop

0x10a4249d0:	vmovdqu	0x10(%rcx,%rdx,4),%xmm0		; B[i:i+3] => xmm0
0x10a4249d6:	vpadd	0x10(%r10,%rdx,4),%xmm0,%xmm0		; C[i:i+3] + xmm0 => xmm0
0x10a4249dd:	vmovdqu	%xmm0,0x10(%r8,%rdx,4)		; xmm0 => A[i:i+3]
0x10a4249e4:	add	\$0x4,%edx		; i += 4
0x10a4249e7:	cmp	%r9d,%edx		;
0x10a4249ea:	j1	0x10a4249d0		; if (i < (MAX-4)) repeat



Manual unrolling?

```
for (long l = 0; l < MAX-4; l+=4) { // main loop
    A[l+0] = B[l+0] + C[l+0];
    A[l+1] = B[l+1] + C[l+1];
    A[l+2] = B[l+2] + C[l+2];
    A[l+3] = B[l+3] + C[l+3];
}
```



Nope... No unrolling during compilation, hence no vectorization.

Vectorized loop

```
int i = 0;
for (; i < MAX-4; i+=4) {           // main loop
    A[i:i+3] = B[i:i+3] + C[i:i+3];
}
for (; i < MAX; i++) {              // post-loop
    A[i] = B[i] + C[i];
}
```

// main-loop

0x10a4249d0: **vmovdqu** 0x10(%rcx,%rdx,4),%xmm00x10a4249d6: **vpaddd** 0x10(%r10,%rdx,4),%xmm0,%xmm00x10a4249dd: **vmovdqu** %xmm0,0x10(%r8,%rdx,4)0x10a4249e4: **add** \$0x4,%edx0x10a4249e7: **cmp** %r9d,%edx0x10a4249ea: **j1** 0x10a4249d0

...

// post-loop

0x10a4249f4: **mov** 0x10(%r10,%rdx,4),%ebx ; A[i] => ebx0x10a4249f9: **add** 0x10(%rcx,%rdx,4),%ebx ; B[i] + ebx => ebx0x10a4249fd: **mov** %ebx,0x10(%r8,%rdx,4) ; ebx => C[i]0x10a424a02: **inc** %edx ; i++0x10a424a04: **cmp** %r11d,%edx ;0x10a424a07: **j1** 0x10a4249f4 ; if (i < MAX) repeat

// ???

0x10a42499d: **mov** 0x10(%r10,%rdx,4),%r9d0x10a4249a2: **add** 0x10(%rcx,%rdx,4),%r9d0x10a4249a7: **mov** %r9d,0x10(%r8,%rdx,4)0x10a4249ac: **inc** %edx0x10a4249ae: **cmp** %edi,%edx0x10a4249b0: **j1** 0x10a42499d

...

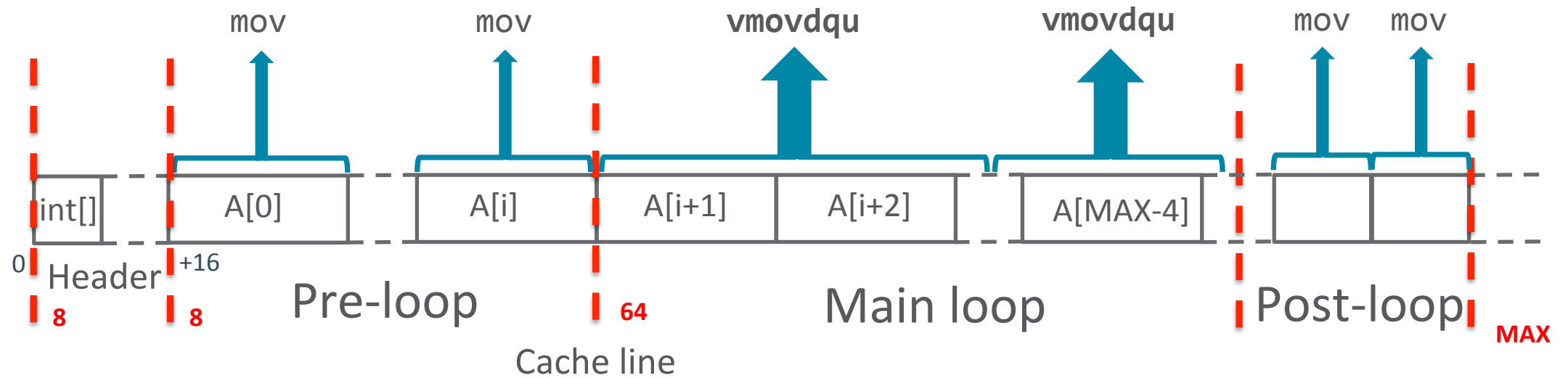
// main-loop

0x10a4249d0: **vmovdqu** 0x10(%rcx,%rdx,4),%xmm00x10a4249d6: **vpaddd** 0x10(%r10,%rdx,4),%xmm0,%xmm00x10a4249dd: **vmovdqu** %xmm0,0x10(%r8,%rdx,4)0x10a4249e4: **add** \$0x4,%edx0x10a4249e7: **cmp** %r9d,%edx0x10a4249ea: **j1** 0x10a4249d0

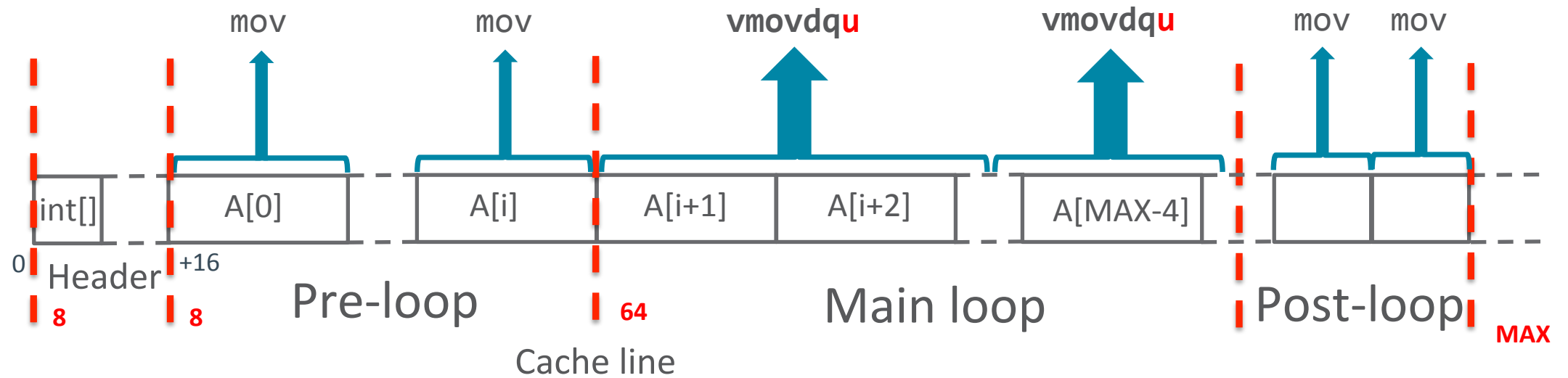
Vectorized loop

```
int i = 0, prefix = ???;  
for (; i < prefix; i++) {           // pre-loop  
    A[i] = B[i] + C[i];  
}  
for (; i < MAX-4; i+=4) {           // main loop  
    A[i:i+3] = B[i:i+3] + C[i:i+3];  
}  
for (; i < MAX; i++) {              // post-loop  
    A[i] = B[i] + C[i];  
}
```

Alignment



Alignment



Core i7, 1x2x2, Haswell (AVX2)
8u121, macos-x64, ns/op

MAX = 1000

T	no unrolling	not vectorized	vectorized
byte	592 ±6	506 ±6	159 ±4
short	541 ±7	495 ±4	140 ±3
char	537 ±4	493 ±4	141 ±2
int	532 ±5	490 ±4	154 ±2
long	533 ±8	492 ±5	157 ±2
float	530 ±4	489 ±7	155 ±2
double	526 ±5	483 ±4	172 ±3

```
<any T> void add (T[] A, T[] B, T[] C) {  
    for (int i = 0; i < MAX; i++) {  
        A[i] = B[i] + C[i];  
    }  
}
```

Main loop

```

0x10a4249d0: vmovdqu 0x10(%rcx,%rdx,4),%xmm0      ; B[i:i+3] => xmm0
0x10a4249d6: vpaddd 0x10(%r10,%rdx,4),%xmm0,%xmm0 ; C[i:i+3] + xmm0 => xmm0
0x10a4249dd: vmovdqu %xmm0,0x10(%r8,%rdx,4)        ; xmm0 => A[i:i+3]

0x10a4249e4: add     $0x4,%edx                     ; i += 4

0x10a4249e7: cmp     %r9d,%edx                     ;
0x10a4249ea: jl      0x10a4249d0                   ; if (i < (MAX-4)) repeat

```

Hmm... Why not 256-bit?

```

0x10a4249d0: vmovdqu 0x10(%rcx,%rdx,4),%xmm0      ; B[i:i+3] => xmm0
0x10a4249d6: vpaddd 0x10(%r10,%rdx,4),%xmm0,%xmm0    ; C[i:i+3] + xmm0 => xmm0
0x10a4249dd: vmovdqu %xmm0,0x10(%r8,%rdx,4)          ; xmm0 => A[i:i+3]

0x10a4249e4: add     $0x4,%edx                       ; i += 4

0x10a4249e7: cmp     %r9d,%edx                       ;
0x10a4249ea: jl      0x10a4249d0                     ; if (i < (MAX-4)) repeat

```

All right, compiler problem. Fixed in 9.

```
→ 0x117023512: vmovdqu 0x10(%rbx,%rcx,4),%ymm0
   0x117023518: vpaddd 0x10(%rdi,%rcx,4),%ymm0,%ymm0
   0x11702351e: vmovdqu %ymm0,0x10(%r9,%rcx,4)
   ...
   0x11702354f: vmovdqu 0x70(%rbx,%r8,4),%ymm0
   0x117023556: vpaddd 0x70(%rdi,%r8,4),%ymm0,%ymm0
   0x11702355d: vmovdqu %ymm0,0x70(%r9,%r8,4)

   0x117023564: add      $0x20,%ecx

   0x117023567: cmp      %r10d,%ecx
   0x11702356a: jl       0x117023512
```

But wait... What happened to main loop?

```

→ 0x117023512: vmovdqu 0x10(%rbx,%rcx,4),%ymm0           ; iteration #1
   0x117023518: vpaddd 0x10(%rdi,%rcx,4),%ymm0,%ymm0           ; A[i:i+7] = B[...] + C[...]
   0x11702351e: vmovdqu %ymm0,0x10(%r9,%rcx,4)                   ;
   ...                                                ; ...
   0x11702354f: vmovdqu 0x70(%rbx,%r8,4),%ymm0           ; iteration #4
   0x117023556: vpaddd 0x70(%rdi,%r8,4),%ymm0,%ymm0           ; A[i+24:i+31] = ...
   0x11702355d: vmovdqu %ymm0,0x70(%r9,%r8,4)                   ;

   0x117023564: add      $0x20,%ecx                       ; i += 32; // (4*8)

   0x117023567: cmp      %r10d,%ecx
   0x11702356a: jl       0x117023512

```



JDK / JDK-8129920

Vectorized loop unrolling

Agile Board

Details

Type:	Enhancement	Status:	RESOLVED
Priority:	P3	Resolution:	Fixed
Affects Version/s:	9	Fix Version/s:	9
Component/s:	hotspot		
Labels:	None		
Environment:	Only applicable for x86 targets. Vendors will need to enable first unrolling queries and possible augment this work for non x86 targets.		
Subcomponent:	compiler		
Resolved In Build:	b75		
CPU:	x86		
OS:	generic		

People

Assignee:



Michael Berg

Reporter:



Michael Berg

Votes:

0

Vote for this issue

Watchers:

2

Start watching this issue

Dates

JDK-8129920: Vectorized Loop Unrolling

“... we leverage unroll factors from the baseline loop which are much larger to obtain **optimum throughput on x86 architectures**. The uplift range on SpecJvm2008 is seen on scimark.lu.{small|large} with uplift noted at **3%** and **8%** respectively. We see as much as **1.5x** uplift on vector centric micros like reductions on default optimizations.”

Michael Berg, Intel

[JDK-8129920](#)

JDK 9: Vectorized Loop: Before Unrolling

```
for (; i < a; i++) { ... }           // pre-loop

for (; i < MAX-4; i+=4) {           // main loop
    A[i:i+3] = B[i:i+3] + C[i:i+3];
}

for (; i < MAX; i++) {               // post-loop
    A[i] = B[i] + C[i];
}
```

JDK 9: Vectorized Loop: After Unrolling

```
for (; i < MAX-step; i+=step) {    // main loop
    A[i:i+V]          = B[i:i+V]      + C[i:i+V];
    A[i+V:i+2*V]      = B[i+V:i+2*V]  + C[i+V:i+2*V];
    A[i+2*V:i+3*V]    = B[i+2*V:i+3*V] + C[i+2*V:i+3*V];
    A[i+3*V:i+4*V]    = B[i+3*V:i+4*V] + C[i+3*V:i+4*V];
}

for (; i < MAX; i++) { A[i] = B[i] + C[i]; } // post-loop

int step = 4 /*unroll_factor*/ * max_vector_size;
int V = max_vector_size - 1;
```

JDK 9: Vectorized Main Loop

```
int step = unroll_factor * max_vector_size;

for (; i < MAX-step; i+=step) { ... }    // main loop

// NB! Up to (unroll_factor * max_vector_size) iterations
for (; i < MAX; i++) { A[i] = B[i] + C[i]; } // post-loop
```

JDK-8149421: Vectorized Post Loops

“the addition of atomic unrolled drain loops which precede fix-up segments and which are **significantly faster** than scalar code. The requirement is that the main loop is super unrolled after vectorization. I see up to **54%** uplift on micro benchmarks on x86 targets for loops which pass superword vectorization and which meet the above criteria.”

Michael Berg, Intel

hotspot-compiler-dev@ojn

JDK 9: Vectorized Post-loop

```
for (; i < MAX-4; i+=4) { // vectorized post-loop  
    A[i:i+3] = B[i:i+3] + C[i:i+3];  
} // trip-count in [0; unroll_factor)
```

```
for (; i < MAX; i++) { // post-loop  
    A[i] = B[i] + C[i];  
} // trip-count in [0; max_vector_size)
```

Core i7, 1x2x2, Haswell (AVX2)
8u121, macos-x64, ns/op

MAX = 1000

T	no unrolling	not vectorized	vectorized	jdk9-b163
byte	592 ±6	506 ±6	159 ±4	69 ±3
short	541 ±7	495 ±4	140 ±3	69 ±4
char	537 ±4	493 ±4	141 ±2	68 ±2
int	532 ±5	490 ±4	154 ±2	74 ±1
long	533 ±8	492 ±5	157 ±2	141 ±1
float	530 ±4	489 ±7	155 ±2	80 ±3
double	526 ±5	483 ±4	172 ±3	167 ±2

```
<any T> void add (T[] A, T[] B, T[] C) {  
    for (int i = 0; i < MAX; i++) {  
        A[i] = B[i] + C[i];  
    }  
}
```

Core i7, 1x2x2, Haswell (AVX2)
macos-x64

T = **int**

MAX	8u121	jdk9-b163	
0	2.0 ±0	2 ±0	ns/op
1	3.8 ±0	3.8 ±0	
10	7.3 ±2	8.2 ±1	
100	22 ±1	17 ±1	
10^3	153 ±4	73 ±3	
10^4	2058 ±57	2025 ±14	µs/op
10^5	36±1	35 ±1	
10^6	858 ±34	883 ±14	
10^7	8751 ±145	9144 ±14	

```
int[] A;  
for (int i = 0; i < MAX; i++)  
    A[i]++;
```

```
int[] A;
for (int i = 0; i < MAX; i++)
    A[i]++;
```



[Constants]

8u121, AVX2 (Haswell)

0x102222b60: 0x00000001

...

vmovq 0x102222b60,%xmm0

vpunpcklqdq %xmm0,%xmm0,%xmm0

vinseriti128 \$0x1,%xmm0,%ymm0,%ymm0

// Main loop

vmovdqu 0x10(%r10,%rcx,4),%ymm1

vpaddd %ymm0,%ymm1,%ymm1

vmovdqu %ymm1,0x10(%r11,%rcx,4)

add \$0x8,%ecx

cmp %r9d,%ecx

j1 ...

```
int[] A;  
for (int i = 0; i < MAX; i++)  
    A[i] *= 10;
```

```
int[] A;  
for (int i = 0; i < MAX; i++)  
    A[i] *= 10;
```



```
// Main loop
```

```
vmovdqu 0x10(%r10,%r11,4),%ymm0
```

```
vpslld $0x1,%ymm0,%ymm1
```

```
vpslld $0x3,%ymm0,%ymm0
```

```
vpadd %ymm0,%ymm1,%ymm0
```

```
vmovdqu %ymm0,0x10(%r10,%r11,4)
```

```
add $0x8,%r11d
```

```
cmp %r8d,%r11d
```

```
j1 ...
```

```
int[] A, B, C;  
for (int i = 0; i < MAX; i++)  
    A[i] = B[i] * C[i];
```

```
int[] A, B, C;  
for (int i = 0; i < MAX; i++)  
    A[i] = B[i] * C[i];
```



```
// Main loop
```

```
vmovdqu 0x10(%rcx,%rdx,4),%xmm0  
vpmulld 0x10(%r10,%rdx,4),%xmm0,%xmm0  
vmovdqu %xmm0,0x10(%r8,%rdx,4)
```

```
add      $0x4,%edx  
cmp      %r9d,%edx  
jl       ...
```

Strided Access

```
float[] A, B, C;  
for (int i = 0; i < MAX; i++)  
    A[i] = B[2*i] * C[2*i];
```

Strided Access

```
float[] A, B, C;  
for (int i = 0; i < MAX; i++)  
    A[i] = B[2*i] * C[2*i];
```

VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

Strided Access

```
int[] A, B, C;
for (int i = 0; i < MAX; i++)
    A[i] = B[2*i] * C[2*i];
```



// Main loop

```
...
vmovss 0x18(%rax,%rcx,4),%xmm4
vaddss 0x18(%rdx,%rcx,4),%xmm4,%xmm1
vmovss %xmm1,0x14(%r11,%r9,4)
vmovss 0x20(%rax,%rcx,4),%xmm4
vaddss 0x20(%rdx,%rcx,4),%xmm4,%xmm1
vmovss %xmm1,0x18(%r11,%r9,4)
vmovss 0x28(%rdx,%rcx,4),%xmm4
vaddss 0x28(%rax,%rcx,4),%xmm4,%xmm1
vmovss %xmm1,0x1c(%r11,%r9,4)

add     $0x4,%r10d
...
```

Strided Access

```
int[] A, B, C;
for (int i = 0; i < MAX; i++)
    A[i] = B[2*i] * C[2*i];
```



VGATHERD* in AVX2, but:

- no scatter operations
- only floating point variants

// Main loop

```
...
vmovss 0x18(%rax,%rcx,4),%xmm4
vaddss 0x18(%rdx,%rcx,4),%xmm4,%xmm1
vmovss %xmm1,0x14(%r11,%r9,4)
vmovss 0x20(%rax,%rcx,4),%xmm4
vaddss 0x20(%rdx,%rcx,4),%xmm4,%xmm1
vmovss %xmm1,0x18(%r11,%r9,4)
vmovss 0x28(%rdx,%rcx,4),%xmm4
vaddss 0x28(%rax,%rcx,4),%xmm4,%xmm1
vmovss %xmm1,0x1c(%r11,%r9,4)

add     $0x4,%r10d
...
```

What about Unsafe?

What about Unsafe?

```
// On-heap
long off = Unsafe.ARRAY_INT_BASE_OFFSET;
for (int i = 0; i < MAX; i++) {
    // A[i] = B[i] + C[i]
    int val = U.getInt(B, off)
              + U.getInt(C, off);
    U.putInt(A, off, val);
    off += Unsafe.ARRAY_INT_INDEX_SCALE;
}
```

What about Unsafe?

```
// On-heap
long off = Unsafe.ARRAY_INT_BASE_OFFSET;
for (int i = 0; i < MAX; i++) {
    // A[i] = B[i] + C[i]
    int val = U.getInt(B, off)
              + U.getInt(C, off);
    U.putInt(A, off, val);
    off += Unsafe.ARRAY_INT_INDEX_SCALE;
}
```



// Main loop

```
...
mov    %rdx,%rdi
add    %r9,%rdi
mov    (%rcx),%r8d
...
add    $0x20,%r9
add    $0x8,%r10d
cmp    %eax,%r10d
jl     ...
```

What about Unsafe?

```
// Off-heap
long addrA = U.allocateMemory(...);
long addrB = U.allocateMemory(...);
long addrC = U.allocateMemory(...);

for (int i = 0; i < MAX; i++) {
    long off = i * 4;
    int val = U.getInt(null, addrB + off)
              + U.getInt(null, addrC + off);
    U.putInt(null, addrA + off, val);
}
```

// Main loop

```
...
mov    0x0(%rbp,%rax,1),%r11d
add    (%rdi,%rax,1),%r11d
mov    %r11d, (%rbx,%rax,1)

...
add     $0x8,%r10d
cmp     %r14d,%r10d
jl      ...
```

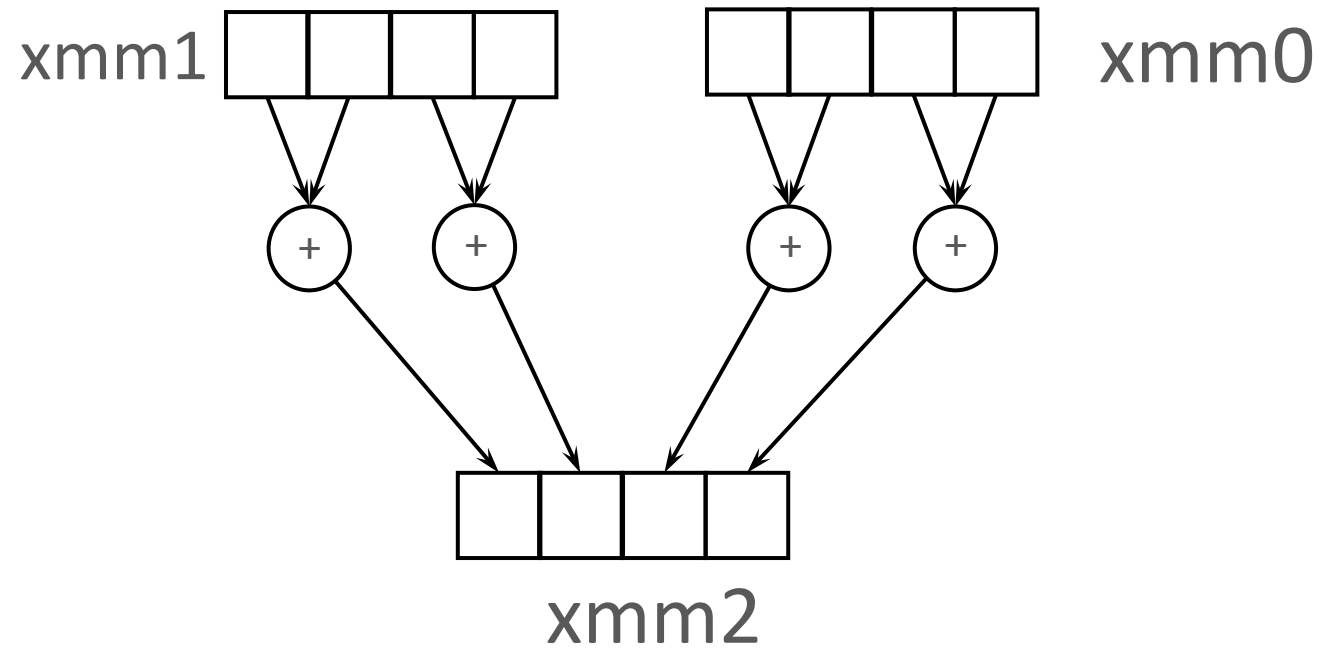


Unsafe == Fast

Unsafe is Fast

Reductions

Horizontal Addition



VPHADDD %xmm0,%xmm1,%xmm2



JDK / JDK-8074981

Integer/FP scalar reduction optimization

Agile Board

Details

Type:  Enhancement

Status: **RESOLVED**

Priority:  P3

Resolution: Fixed

Affects Version/s: 9

Fix Version/s: 9

Component/s: hotspot

Labels: None

Subcomponent: compiler

Resolved In Build: b64

```
public int sum(int[] A) {  
    int sum = 0;  
    for (int a : A) {  
        sum += a;  
    }  
    return sum;  
}
```

```

public int sum(int[] A) {
    int sum = 0;
    for (int a : A) {
        sum += a;
    }
    return sum;
}

```



```
// Main loop
```

```

add 0x10(%r8,%rcx,4),%eax
add 0x14(%r8,%rcx,4),%eax
add 0x18(%r8,%rcx,4),%eax
add 0x1c(%r8,%rcx,4),%eax
add 0x20(%r8,%rcx,4),%eax
add 0x24(%r8,%rcx,4),%eax
add 0x28(%r8,%rcx,4),%eax
add 0x2c(%r8,%rcx,4),%eax

```

```

add $0x8,%ecx
cmp %r10d,%ecx
jl  ...

```

“Reduction vector optimization could be **expensive for simple expressions** because it uses several additional instructions per vector. [...] We need to restrict reduction optimization only to cases when it is beneficial.”

[8074981: Restrict reduction optimization](#)

```
int dotProduct(int[] A, int[] B) {  
    int r = 0;  
    for (int i = 0; i < MAX; i++) {  
        r += A[i]*B[i];  
    }  
    return r;  
}
```

```

int dotProduct(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < MAX; i++) {
        r += A[i]*B[i];
    }
    return r;
}

```



```

// Vectorized post-loop
vmovdqu 0x10(%rdi,%r11,4),%ymm0
vmovdqu 0x10(%rbx,%r11,4),%ymm1
vpmulld %ymm0,%ymm1,%ymm0
vphaddb %ymm0,%ymm0,%ymm3
vphaddb %ymm1,%ymm3,%ymm3
vextracti128 $0x1,%ymm3,%xmm1
vpaddb %xmm1,%xmm3,%xmm3
vmovd %eax,%xmm1
vpaddb %xmm3,%xmm1,%xmm1
vmovd %xmm1,%eax
add $0x8,%r11d
cmp %r8d,%r11d
jl 0x117e23668

```

```

int dotProduct(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < MAX; i++) {
        r += A[i]*B[i];
    }
    return r;
}

```

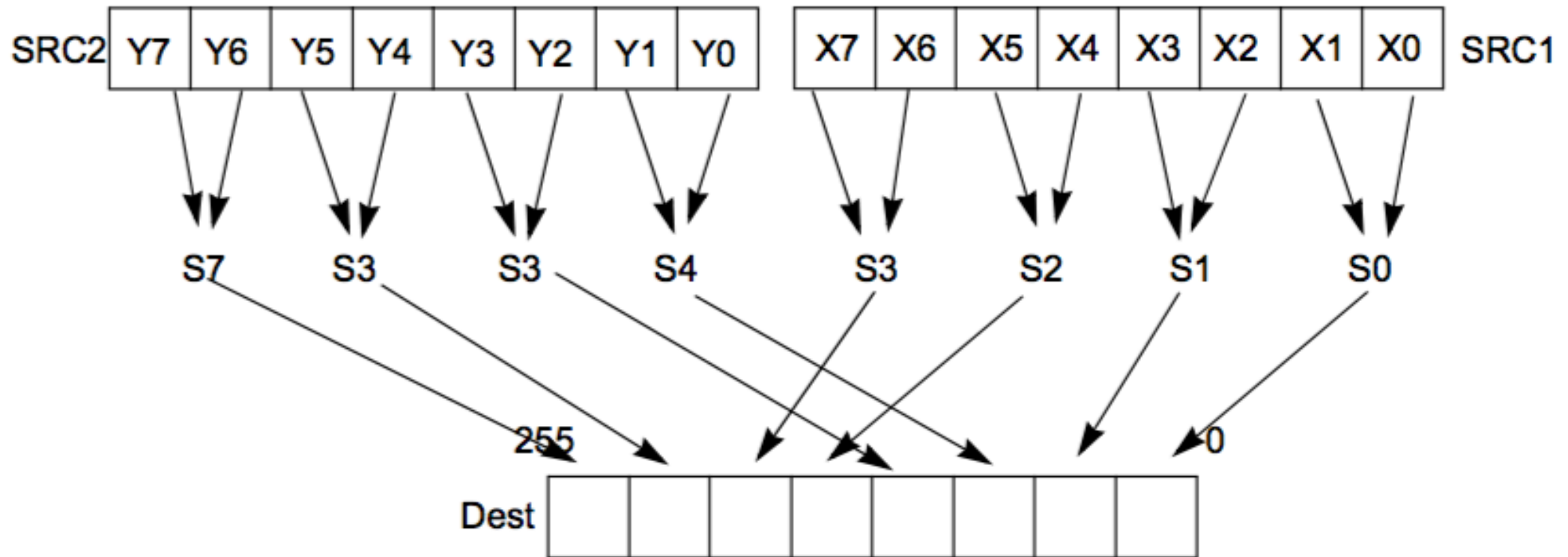


```

// Vectorized post-loop
vmovdqu 0x10(%rdi,%r11,4),%ymm0
vmovdqu 0x10(%rbx,%r11,4),%ymm1
vpmulld %ymm0,%ymm1,%ymm0
vphadd %ymm0,%ymm0,%ymm3
vphadd %ymm1,%ymm3,%ymm3
vextracti128 $0x1,%ymm3,%xmm1
vpadd %xmm1,%xmm3,%xmm3
vmovd %eax,%xmm1
vpadd %xmm3,%xmm1,%xmm1
vmovd %xmm1,%eax
add $0x8,%r11d
cmp %r8d,%r11d
jl 0x117e23668

```

VPHADDD



Fused Multiply-Add (FMA)

Fused Operations

- Single Instruction – Multiple Nested operations

- Use cases

- dot product

```
for (int i = 0; i < MAX; i++)  
    r = r + A[i]*B[i];
```

- matrix multiplication

```
for (int k = 0; k < MAX; k++)  
    r = r + A[i][k]*B[k][j];
```

FMA4 (only AMD)

Mnemonic	Operands	Operation
VFMADDPDy	ymm, ymm, ymm/m256	a = b * c + d
VFMADDPsy		
VFMADDPDx	xmm, xmm, xmm/m128	
VFMADDPsx		
VFMADDSD	xmm, xmm, xmm/m64	
VFMADDSS	xmm, xmm, xmm/m32	

FMA3 (both Intel & AMD)

Mnemonic	Operation
VFMADD132...	$a = a * c + b$
VFMADD213...	$a = b * a + c$
VFMADD231...	$a = b * c + a$

```
float[] A, B, C, D = ...;
```

```
for (int i = 0; i < MAX; i++) {
    A[i] = B[i] * C[i] + D[i];
}
```



Vectorized, no FMA.

```
// Vectorized post-loop
```

```
vmovdqu 0x10(%r8,%r11,4),%ymm0
```

```
vmulps 0x10(%rcx,%r11,4),%ymm0,%ymm0
```

```
vaddps 0x10(%rax,%r11,4),%ymm0,%ymm0
```

```
vmovdqu %ymm0,0x10(%rdx,%r11,4)
```

```
add      $0x8,%r11d
```

```
cmp      %r10d,%r11d
```

```
j1      ...
```

Math.fma() // @since 9

```
float[] A, B, C, D = ...;
```

```
for (int i = 0; i < MAX; i++) {
    A[i] = Math.fma(B[i], C[i], D[i]);
}
```



Not vectorized, scalar FMA.

```
// Post-loop
```

```
vmovss 0x10(%r11,%rbx,4),%xmm1
vmovss 0x10(%r9,%rbx,4),%xmm0
vmovss 0x10(%r8,%rbx,4),%xmm3
vfmadd231ss %xmm1,%xmm0,%xmm3
vmovss %xmm3,0x10(%r10,%rbx,4)
```

```
inc     %ebx
cmp     %edi,%ebx
jl      ...
```

JDK 9: Other Enhancements in SuperWord

8153998: Masked vector post loops

8080325: SuperWord loop unrolling analysis

8151573: Multiversioning for range check elimination

8135028: support for vectorizing double precision sqrt

8076284: Improve vectorization of parallel streams

8139340: SuperWord enhancement to support vector conditional move (CMovVD) on Intel AVX cpu

What else in AVX-512?

- Masked vector operations

```
if (B[i] > 0) A[i] = B[i];
```

- Scatter/Gather

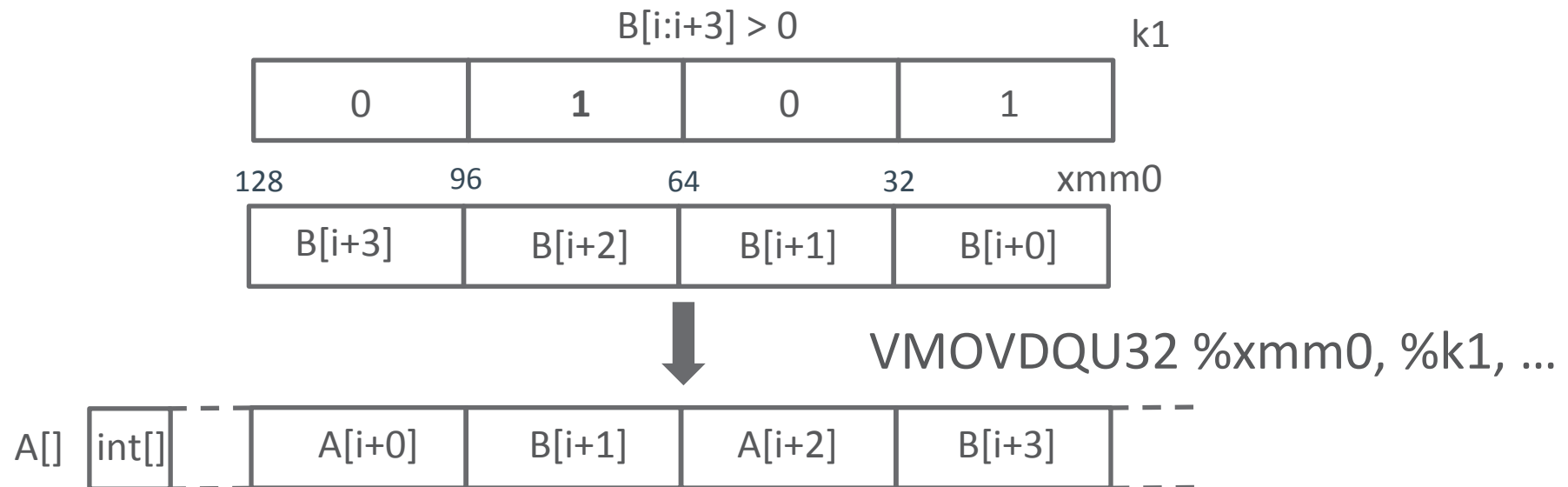
```
A[i] = B[i] + C[D[i]]; // indirect access
```

- Conflict Detection

```
A[B[i]]++; // histogram
```

If-conversion

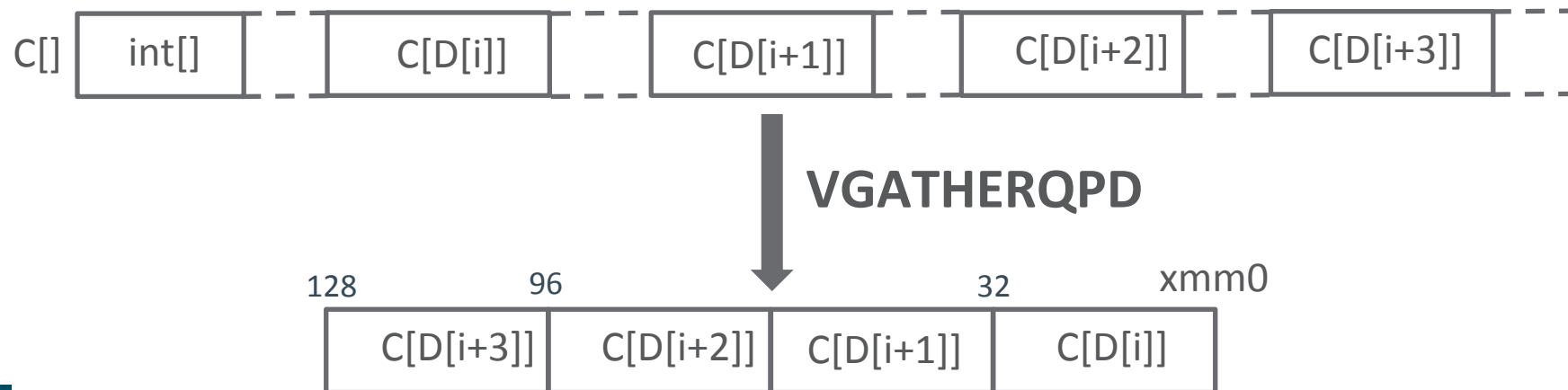
```
for (int i = 0; i < MAX; i++)  
    if (B[i] > 0) {  
        A[i] = B[i];  
    }
```



Indirect Access

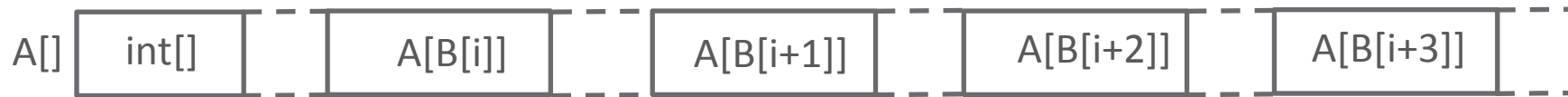
```
for (int i = 0; i < MAX; i++)  
    A[i] = B[i] + C[D[i]];
```

“Gatherers”: fetch data elements using vector-index memory addressing.



Histogram

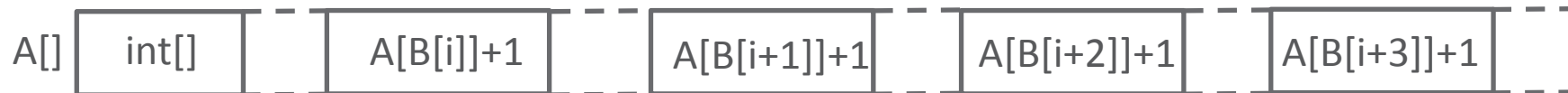
```
for (int i = 0; i < MAX; i++)  
    A[B[i]]++;
```



VPGATHERDD

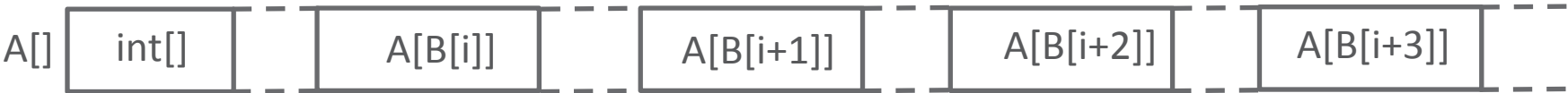


VPSCATTERDD

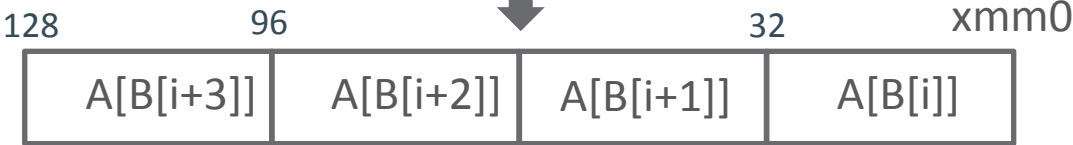


Histogram

```
for (int i = 0; i < MAX; i++)  
    A[B[i]]++;
```

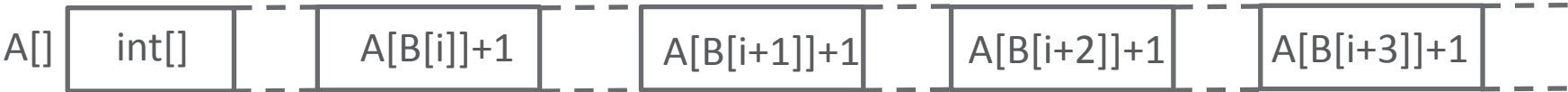


VPGATHERDD



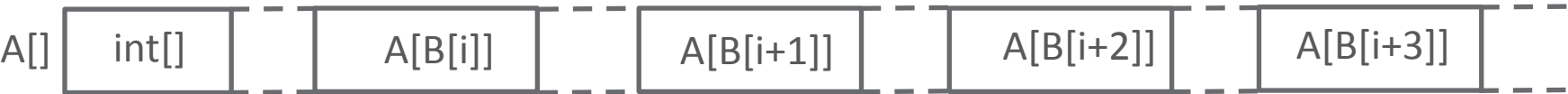
B[i] == B[j]?

VPSCATTERDD



Histogram

```
for (int i = 0; i < MAX; i++)  
    A[B[i]]++;
```



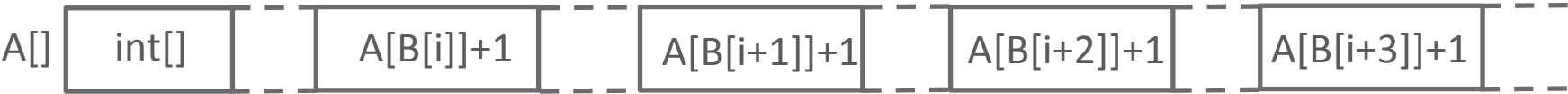
VPGATHERDD



Conflict detection!

VPCONFLICTD

VPSCATTERDD



Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☐ KNC
- ☐ SVML
- ☐ Other

[?](#)

<code>__m128i _mm_abs_epi16 (__m128i a)</code>	pabsw
<code>__m128i _mm_mask_abs_epi16 (__m128i src, __mmask8 k, __m128i a)</code>	vpabsw
<code>__m128i _mm_maskz_abs_epi16 (__mmask8 k, __m128i a)</code>	vpabsw
<code>__m256i _mm256_abs_epi16 (__m256i a)</code>	vpabsw
<code>__m256i _mm256_mask_abs_epi16 (__m256i src, __mmask16 k, __m256i a)</code>	vpabsw
<code>__m256i _mm256_maskz_abs_epi16 (__mmask16 k, __m256i a)</code>	vpabsw
<code>__m512i _mm512_abs_epi16 (__m512i a)</code>	vpabsw
<code>__m512i _mm512_mask_abs_epi16 (__m512i src, __mmask32 k, __m512i a)</code>	vpabsw
<code>__m512i _mm512_maskz_abs_epi16 (__mmask32 k, __m512i a)</code>	vpabsw
<code>__m128i _mm_abs_epi32 (__m128i a)</code>	pabsd
<code>__m128i _mm_mask_abs_epi32 (__m128i src, __mmask8 k, __m128i a)</code>	vpabsd
<code>__m128i _mm_maskz_abs_epi32 (__mmask8 k, __m128i a)</code>	vpabsd
<code>__m256i _mm256_abs_epi32 (__m256i a)</code>	vpabsd
<code>__m256i _mm256_mask_abs_epi32 (__m256i src, __mmask8 k, __m256i a)</code>	vpabsd
<code>__m256i _mm256_maskz_abs_epi32 (__mmask8 k, __m256i a)</code>	vpabsd

Vector ISA Extensions

- 100s of vector instructions on x86
- Intel intrinsic instructions
 - MMX: ~120
 - SSE: ~130
 - SSE2/3/SSSE3/4.1/4.2: ~260
 - AVX/AVX2: ~380

Vector ISA Extensions

- 1000s of vector instructions on x86
- Intel intrinsic instructions
 - MMX: ~120
 - SSE: ~130
 - SSE2/3/SSSE3/4.1/4.2: ~260
 - AVX/AVX2: ~380
 - AVX-512: ~3800



John Rose

@JohnRose00



Following

AVX is the C++ of ISAs. (...That Lovecraftian sense of baffled, fascinated revulsion which grows as you delve for its nether secrets.)

RETWEETS

3

LIKES

3



10:16 PM - 19 Oct 2015

Use Case: UTF-8 <=> UTF-16

	UTF-8	UTF-16
ASCII (1 byte)	0aaaaaaa	00000000 0aaaaaaa
Basic Multilingual Plane (2 or 3 bytes)	110bbbb 10aaaaaa	00000bbb bbaaaaaa
	1110cccc 10bbbbbb 10aaaaaa	ccccbbbb bbaaaaaa
Supplementary Planes (4 bytes)	11110ddd 10ddcccc 10bbbbbb 10aaaaaa uuuu = dddd - 1	110110uu uuccccbb 110111bb bbaaaaaa

Use Case: UTF-8 $\Leftrightarrow$ UTF-16

A Case Study in SIMD Text Processing with Parallel Bit Streams

UTF-8 to UTF-16 Transcoding

Robert D. Cameron

School of Computing Science, Simon Fraser University
cameron@cs.sfu.ca

Java and SIMD today

- Superword optimizations can be very brittle
 - doesn't (and can't) cover all the use cases
- Intrinsics are point fixes, not general
 - powerful, lightweight, and flexible
 - high development costs
- JNI is hard to develop and maintain
 - interoperability overhead between Java & native code
 - CPU dispatching is required

Vector API

Embrace explicit vectorization

Project Panama



Safe Harbor Statement

The following is intended for information purposes only and is not a commitment to deliver any product, feature, or functionality described or suggested herein. It is intended for information purposes only and is not a contract. It is not a commitment to deliver any product, feature, or functionality described or suggested herein. It should not be relied upon in making purchasing decisions. The delivery of any features or functionality described herein is at the discretion of Oracle.



Motivation

Expose data-parallel operations
through a cross-platform API

Motivation

```
Int8Vector x = ..., y = ...;  // vectors of 8 ints  
Int8Vector z = x.add(y);      // element-wise addition
```



```
vpaddq %ymm1,%ymm0,%ymm0
```

Goals

- Maximally expressive and portable API
 - “principle of least astonishment”
 - uniform coverage operations and data types
 - type-safe
- Performant
 - High quality of generated code
 - Competitive with existing facilities for auto-vectorization
- Graceful performance degradation
 - fallback for "holes" in native architectures

Current Status

- Draft API proposed by John Rose
 - Immutable Vector type
 - parameterized by element type & size (Vector<E,S>)

```
Vector<Integer, S256Bit> x = ..., y = ...; // vectors of 8 ints  
Vector<Integer, S256Bit> z = x.add(y);      // element-wise addition
```

- Prototype Implementation in Panama
 - int, float, long, and double elements supported
 - Int128Vector, Int256Vector, ...
 - based on Machine Code Snippets & “super-longs” (Long2, Long4, Long8)

Raw Vectors

- java.lang.Long2 / Long4 / Long8 / ...
 - represent 128/256/512-bit values
- “well-known” to the JVM
 - special treatment in the JVM
 - C2 knows how to map the values to appropriate vector registers

```
// 128-bit vector.  
public /* value */ final class Long2 {  
    private final long l1, l2; // FIXME  
  
    private Long2() { throw new Error(); }  
  
    @HotSpotIntrinsicCandidate  
    public static native Long2 make(long lo, long hi);  
  
    @HotSpotIntrinsicCandidate  
    public native long extract(int i);  
  
    @HotSpotIntrinsicCandidate  
    public boolean equals(Long2 v) { ... }  
  
    ...  
}
```

JVM vs Hardware: Impedance Mismatch

size (bits)	8	16	32	64	128	256	512	...
x86 regs	AL	AX	EAX	RAX	XMM0	YMM0	ZMM0	-
JVM	B	S	I	J	j.l.Long2	j.l.Long4	j.l.Long8	...

Vector Box Elimination

- Optimize away vector boxes in the code
 - required for mapping Vector instances to vector registers in generated code
 - `Vector<Integer,S256Bits> => vector register (ymm) on x86/AVX`
 - crucial for decent performance
- Escape Analysis in C2
 - doesn't cover all the cases (e.g., non-trivial control flow)
 - brittle (depends on inlining decisions; easy for a user to leak an instance)

Vector box elimination

- Value Types (Project Valhalla) for the rescue!
 - represent super-longs & typed vectors as value types
 - let the JIT-compiler do the rest
- Minimal Value Types, as the first step
 - <http://cr.openjdk.java.net/~jrose/values/shady-values.html>

Valhalla JVM vs Hardware

size (bits)	8	16	32	64	128	256	512	...
x86 regs	AL	AX	EAX	RAX	XMM0	YMM0	ZMM0	-
JVM	B	S	I	J	j.l.Long2	j.l.Long4	j.l.Long8	...

Summary

Summary

- SIMD ISA extensions
 - very irregular on x86
 - hard to utilize in cross-platform manner
- JVM
 - auto-vectorization
 - brittle
 - can't cover all the cases
 - intrinsics
 - pros: powerful, lightweight, and flexible
 - cons: point fixes, high development costs

Summary: Future

- JDK 9
 - enhancements in auto-vectorization
 - partial AVX-512 support, code shape improvements on x86
 - new methods and intrinsics
 - `Math.fma()`, `Arrays.vectorizedMismatch()`
- Vector API
 - easy & reliable way to write performant vectorized code
 - work in progress!
 - under active development in Project Panama

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Materials

- Vector interface: <http://cr.openjdk.java.net/~jrose/arrays/vector/Vector.java>
- Prototype: <http://hg.openjdk.java.net/panama/panama/jdk/file/tip/test/panama/vector-api-patchable>
- Minimal Value Types: <http://cr.openjdk.java.net/~jrose/values/shady-values.html>
- Super-longs:
<http://hg.openjdk.java.net/panama/panama/jdk/file/0243d8ef6bd1/src/java.base/share/classes/java/lang/Long2.java>
- Machine Code Snippets: http://cr.openjdk.java.net/~vlivanov/talks/2016_JVMLS_MachineCodeSnippets.pdf

Thank you!

vladimir.x.ivanov@oracle.com
@iwan0www