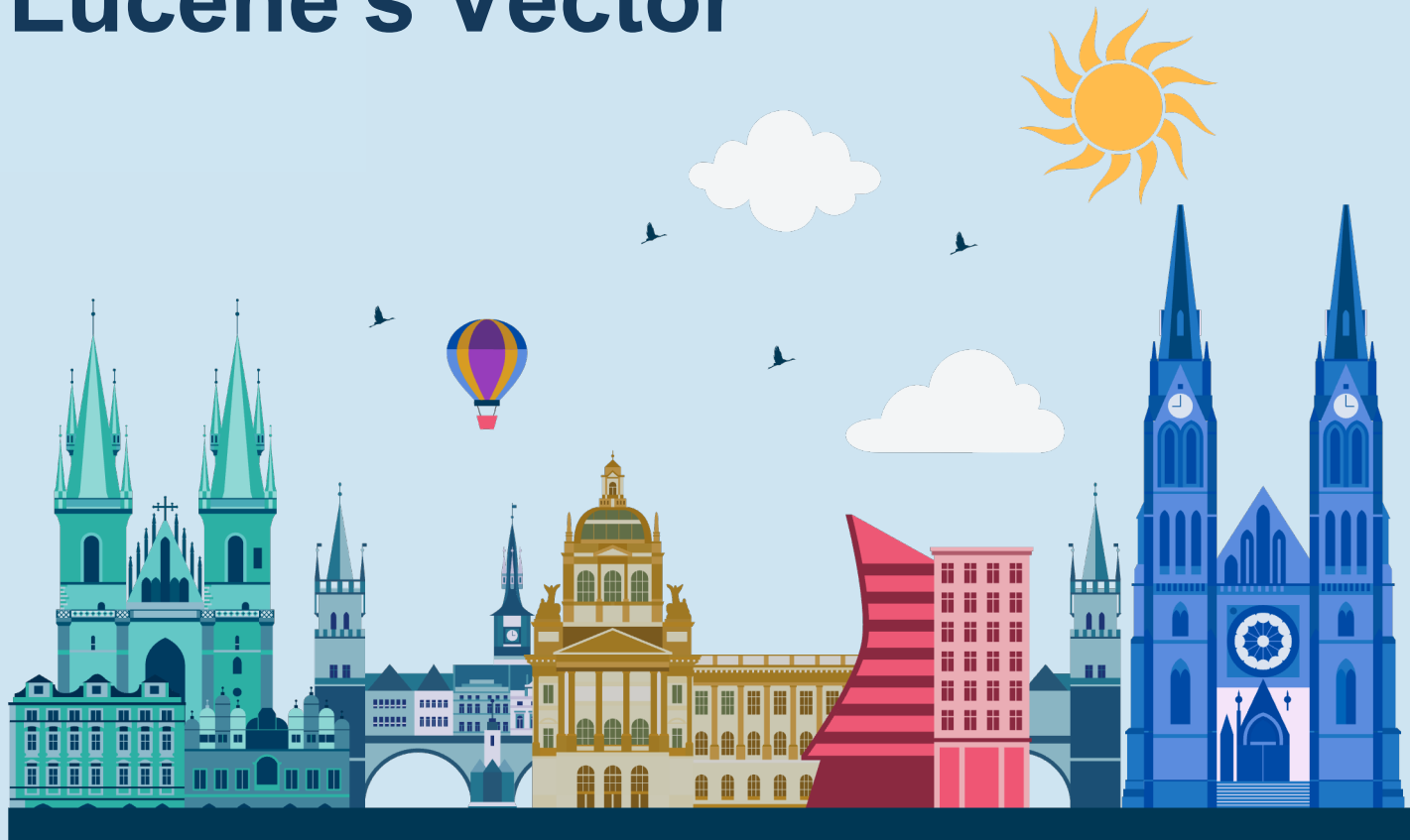
OpenSearchCon EUROPE

Using Native CPU Vectorized Instructions for 60% Speedup on Lucene's Vector Search

Shubham Chaudhary



About Me

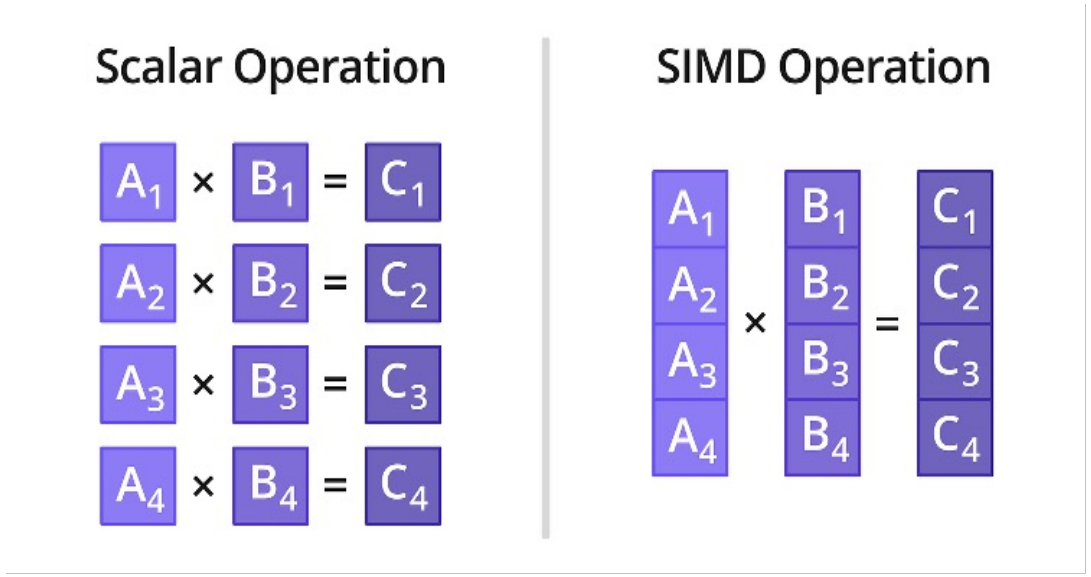
- Software Engineer, Core Search at Amazon
- Apache Lucene Committer

Agenda

- **What is SIMD**
- **Why SIMD matters to vectors search engines**
- **The ARM Gap: JVM Falls Short**
- **The Real Problem: Missing SDOT/UDOT**
- **What is NEON and SVE extensions**
- **Our Solution: Native Vectorization**
  - Compile Time: Multi-Architecture Shared Binary
  - Runtime: Automatic Best-Path Selection
  - Key Benefits
- **Performance Impact: JMH Benchmarks**
- **Memory copy overhead**
- **Using Linker.Option.critical**
- **Results from Prod**
- **Apache Lucene's approach to SIMD**
- **Lucene 11 : Native SIMD support in Apache Lucene**
- **Benefits to OpenSearch**

What is SIMD

- **SIMD** stands for **Single Instruction, Multiple Data**, a type of parallel computing where a single CPU instruction performs the same operation on multiple data points simultaneously.



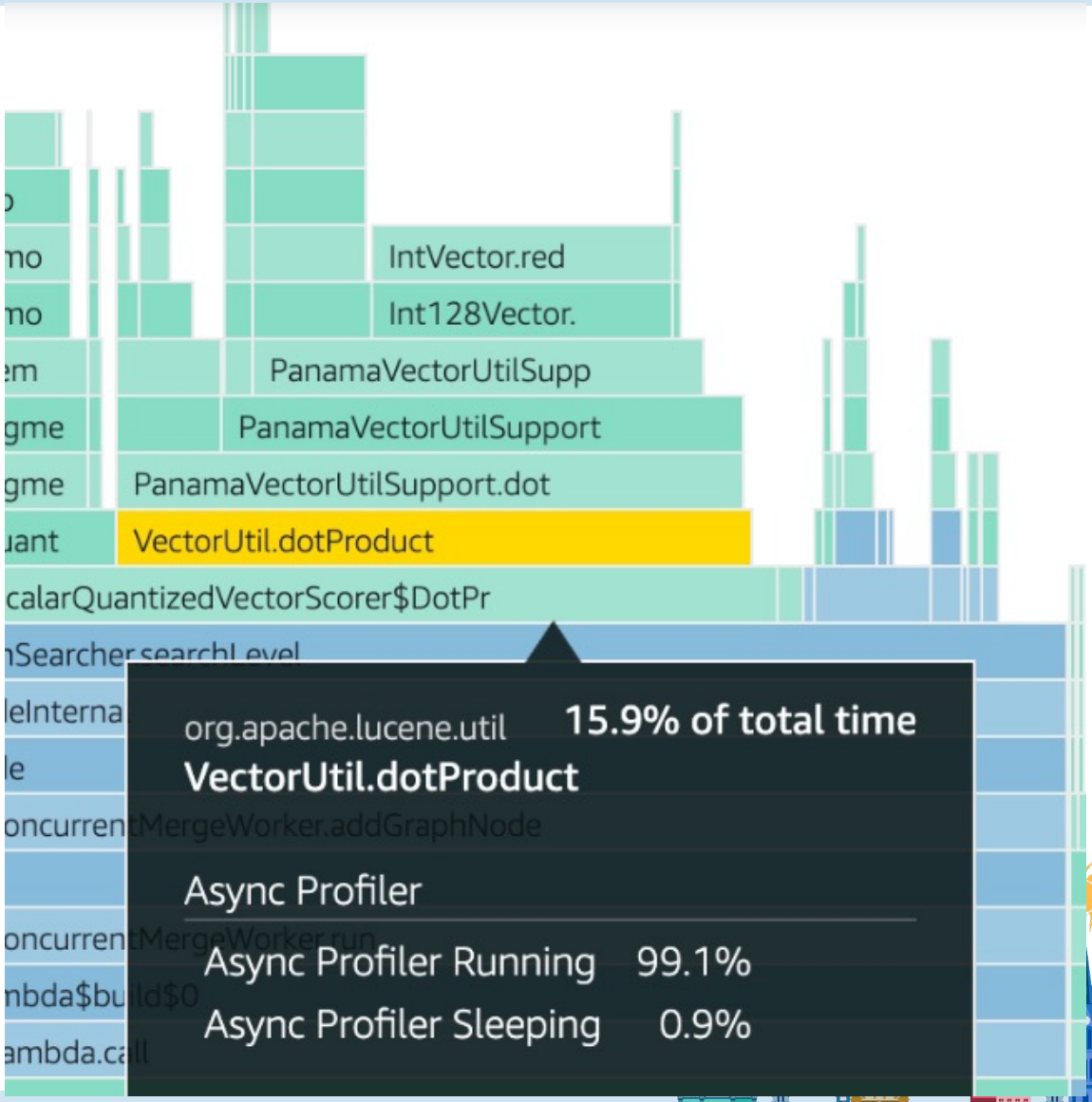
Without SIMD (Scalar)	With SIMD
A CPU would perform one multiplication at a time, sequentially, for each pair of numbers.	Use **one** SIMD "multiply" instruction to multiply all four pairs of numbers at once.
Result: 4 multiplications in 4 single steps	**Result: 4 multiplications in 1 single step instead of four separate steps.**

Why SIMD matter to vectors search engines

OpenSearchCon EUROPE

Dot product = #1 CPU bottleneck in vector search

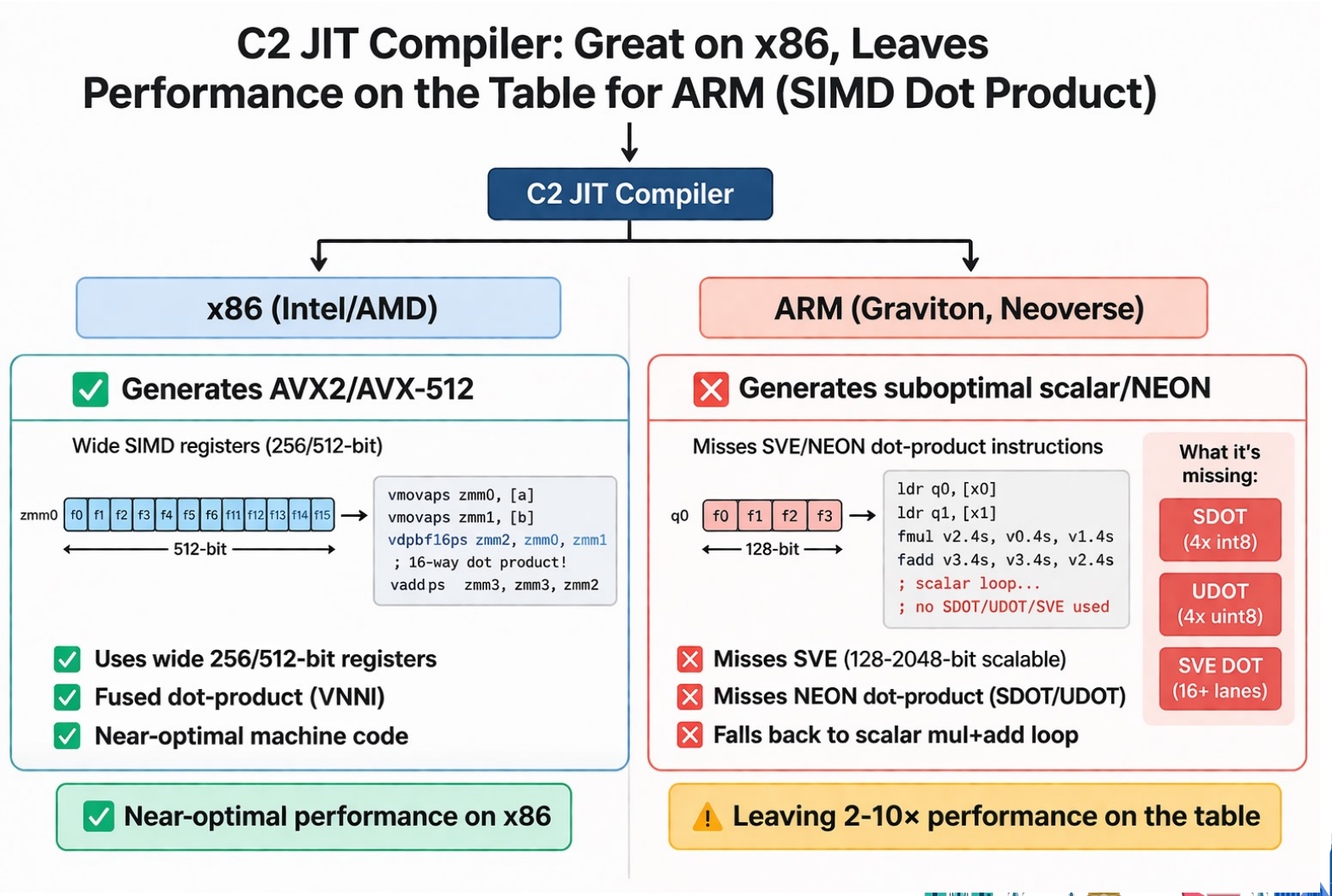
- Billions of products in catalog means every query → thousands of vector comparisons
- High-dimensional embeddings makes it more problematic
- Taking **~15-30%** of CPU time in KNN search path



The ARM Gap: JVM Falls Short

Lucene Already Uses SIMD... So What's Wrong?

We already have SIMD in Lucene(Panama Vector API). The problem isn't the Java code - it's what the JVM does with it on ARM.



The Real Problem: Missing SDOT/UDOT

SDOT/UDOT = Signed/Unsigned Dot Product instruction

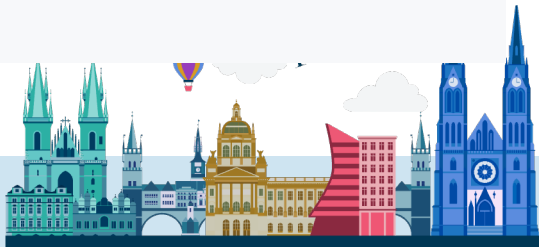
- A single ARM instruction that computes the dot product of four int8 pairs and accumulates into int32.

SDOT → signed int8 (values -128 to 127)
UDOT → unsigned int8 (values 0 to 255)

- It lives in TWO ARM SIMD extensions : **NEON** and **SVE**

Luceneutil PR : https://github.com/mikemccand/luceneutil/pull/423

```
// 7 bit; found matches

>> PAGER=cat perf annotate -i perf2.data -f --stdio | head -100000 | grep "sdot"
    0.00 :   79c:    sdot    v0.4s, v2.16b, v4.16b
    0.00 :   7a8:    sdot    v7.4s, v6.16b, v5.16b
    0.00 :   7b0:    sdot    v22.4s, v3.16b, v16.16b
    0.00 :   7bc:    sdot    v21.4s, v18.16b, v17.16b
    0.28 :   7e4:    sdot    v0.4s, v19.16b, v20.16b
    0.55 :   7ec:    sdot    v7.4s, v23.16b, v30.16b
    0.44 :   7f4:    sdot    v22.4s, v24.16b, v31.16b
    1.81 :   7fc:    sdot    v21.4s, v25.16b, v1.16b
    0.34 :   804:    sdot    v0.4s, v26.16b, v2.16b
    0.58 :   80c:    sdot    v7.4s, v27.16b, v4.16b
    0.61 :   814:    sdot    v22.4s, v28.16b, v5.16b
    1.80 :   81c:    sdot    v21.4s, v29.16b, v3.16b
    0.00 :   8b4:    sdot    v0.4s, v22.16b, v16.16b
    0.00 :   8c0:    sdot    v0.4s, v17.16b, v18.16b
    0.00 :   8cc:    sdot    v0.4s, v19.16b, v23.16b
    0.00 :   8d8:    sdot    v0.4s, v24.16b, v25.16b
```

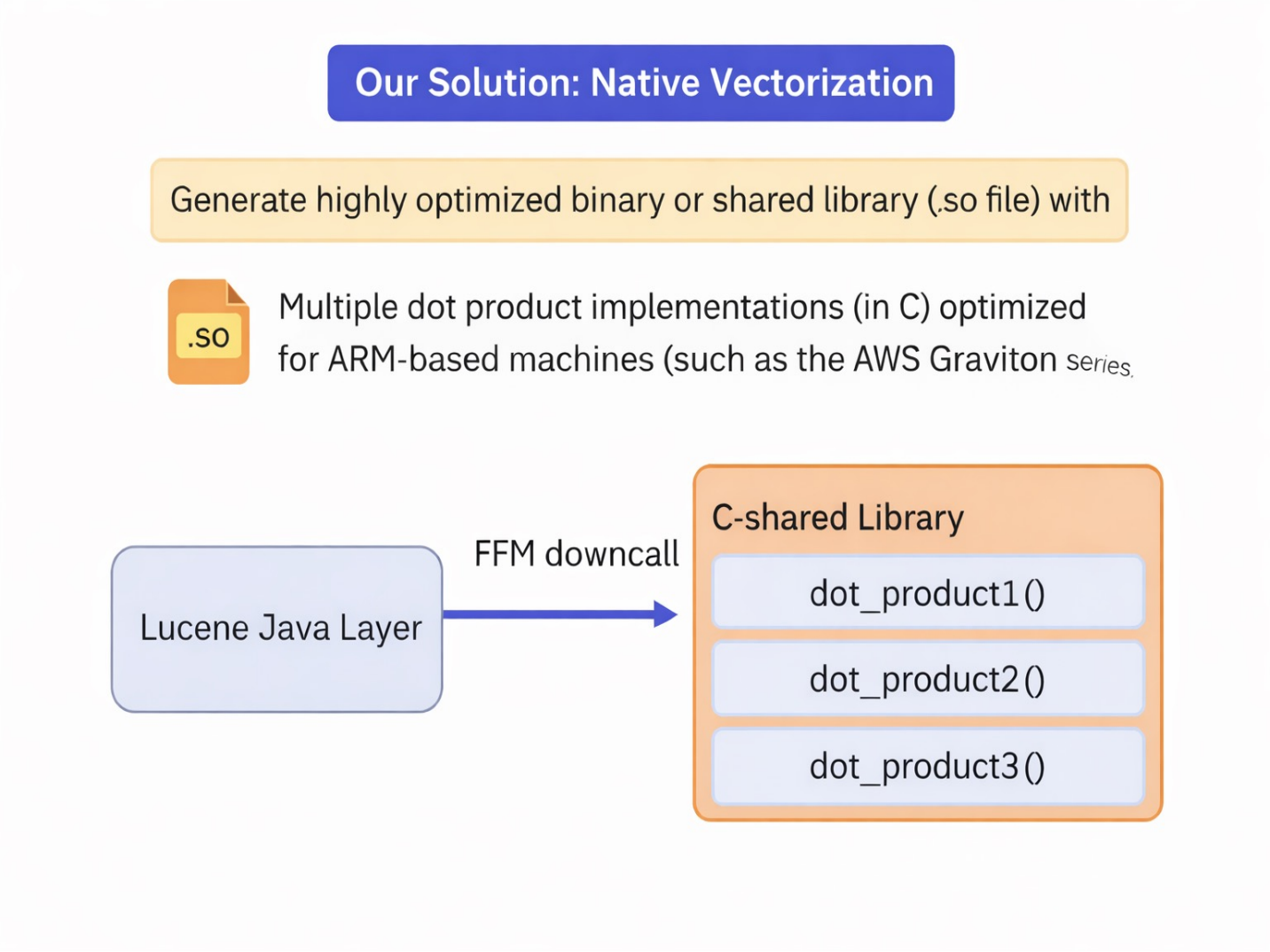


What is NEON and SVE extensions

- **NEON** and **SVE** are ARM ISA(Instruction Set Architecture) vector extensions that provide hardware vector registers and instructions for data-parallel execution.
- Both have SDOT/UDOT - but SVE processes MORE data per instruction due to wider/scalable registers.
- The JVM's C2 compiler uses NEITHER on ARM.

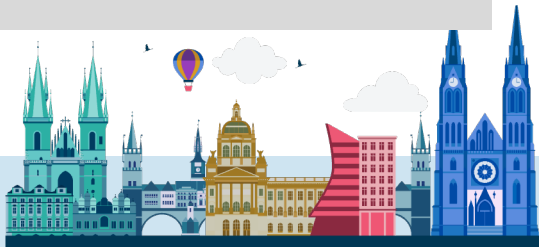
Feature	NEON (ASIMD)	SVE (Scalable Vector Extension)
Introduced	ARMv7 (extended in ARMv8)	ARMv8.2-A (optional)
Vector Size	Fixed 128-bit	Scalable (128–2048 bits)
Programming Model	Fixed-width SIMD	Vector-length agnostic (VLA)
Parallelism	Fixed 128-bit vector width	Scales with hardware width
Dot Product Ops	SDOT / UDOT (int8, optional)	SDOT / UDOT + scalable vectors + predication
Flexibility	Lower	High
Typical Usage	General SIMD (mobile, server, embedded)	HPC, ML, high-performance workloads

Our Solution: Native Vectorization



- Generate **highly optimized binary or shared library(.so file)** with
- Multiple **dot product implementations(in C)** optimized for ARM-based machines (such as the AWS Graviton series)
- Call the native code from Lucene's java layer via **FFM downcall**.

Initial idea on upstream Apache Lucene PR (by Ankur Goel : @goankur) : https://github.com/apache/lucene/pull/13572



Compile Time: Multi-Architecture Shared Binary

COMPILE TIME — BUILD

dotProduct.c

Three implementations with GCC function attributes

▲ SVE · NEON · Scalar

↓

GCC compiler

```
gcc -march=armv8.4-a+dotprod+sve \
  -O3 -fPIC -shared \
  -o libdotProduct.so dot.c
```

Flags: **-march · -O3 · -fPIC · -shared**

Requires **GCC ≥ 10**

↓

libdotProduct.so

One binary — all implementations + runtime CPU selector

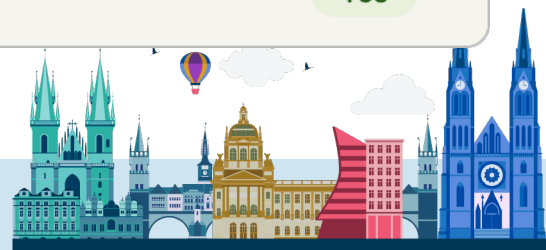
SVE · NEON · Scalar · Runtime selector

- **Goal:** One binary (.so file) that contains ALL implementations
- Compile all 3 C implementations having function attributes into one binary(GCC >= 10)

```
>>  gcc -march=armv8.4-a+dotprod+sve  -shared -o libdotProduct.so dot.c
```

IMPLEMENTATIONS INSIDE THE BINARY

Implementation	ARM ISA required	Intrinsics used	In binary
▲ SVE fastest	armv8.4-a+sve	svdot_s32, svld1_s8	Yes
● NEON fast	arm8.2-a+dotprod	vdotq_s32, vld1q_s8	Yes
■ Scalar fallback	Always included	Basic loop	Yes

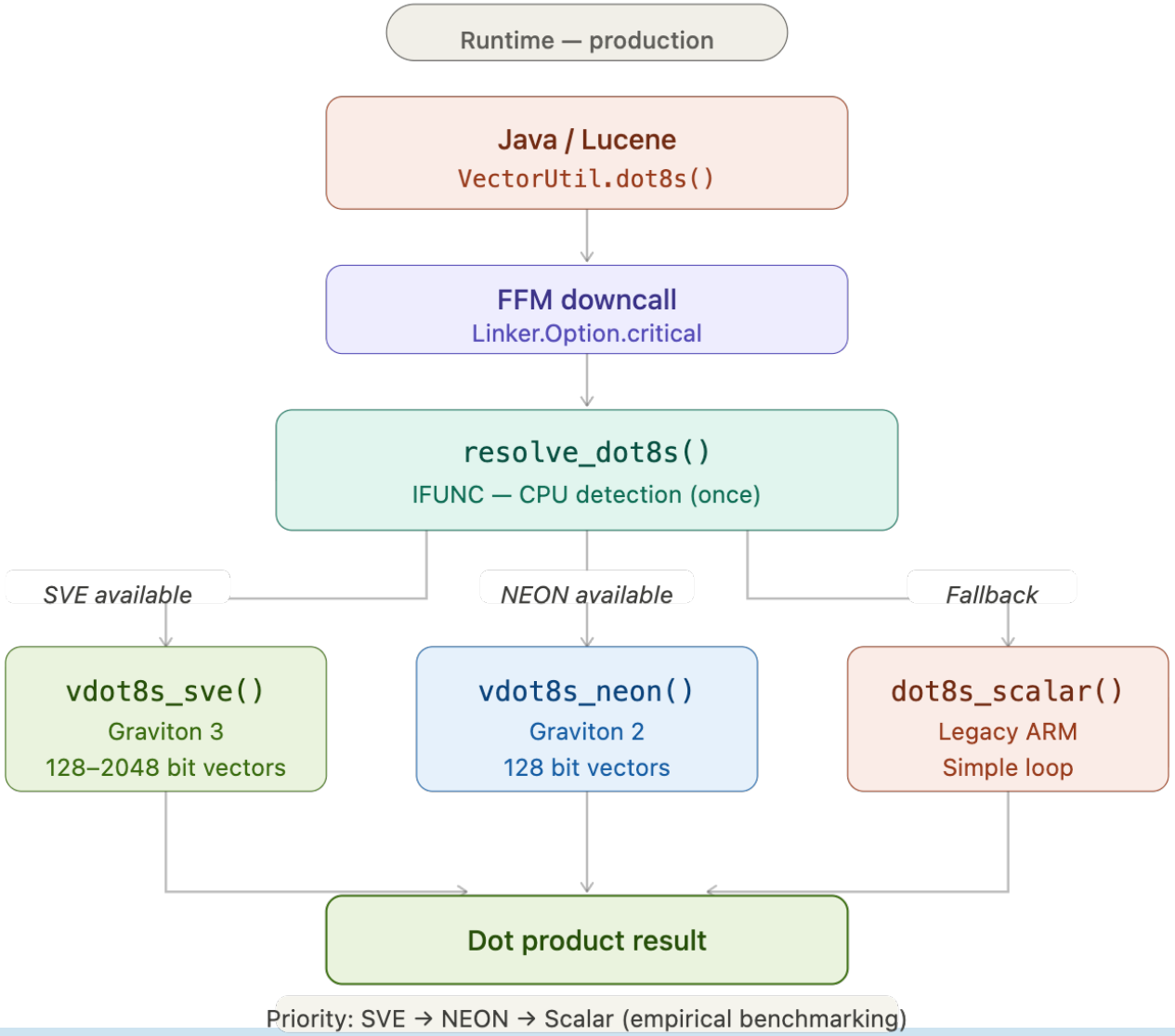


Compile Time: Multi-Architecture Shared Binary


```
#if (defined(__ARM_FEATURE_SVE))
#include <arm_sve.h>

__attribute__((target("arch=armv8.4-a+sve")))
int32_t vdot8s_sve(int8_t vec1[], int8_t vec2[], int32_t limit) {
    // SVE intrinsics code
}
#endif

#if defined(__ARM_NEON)
#include <arm_neon.h>

__attribute__((target("arch=armv8.2-a+dotprod")))
int32_t vdot8s_neon(int8_t vec1[], int8_t vec2[], int32_t limit) {
    // NEON intrinsics code
}
#endif
#endif

int32_t vdot8s_scalar(int8_t vec1[], int8_t vec2[], int32_t limit) {
    // Scalar fallback
    int32_t result = 0;
    for (int32_t i = 0; i < limit; i++) {
        result += vec1[i] * vec2[i];
    }
    return result;
}
```

ALL IMPLEMENTATIONS

Runtime: Automatic Best-Path Selection



- Using **IFUNC(Indirect functions)** to select the optimal implementation at runtime
- Dynamically resolves to the best implementation based on available **hardware capabilities** and the chosen **priority order**
- **Priority Order** (determined empirically via offline benchmarking):

  SVE → NEON → Scalar

Runtime: Automatic Best-Path Selection


```
// Resolve implementation or function
static void* resolve_dot8s(void) {
#ifdef AT_HWCAP
    unsigned long hwcap = getauxval(AT_HWCAP);
#endif
#if defined(HWCAP_SVE) && defined(__ARM_FEATURE_SVE)
    if (hwcap & HWCAP_SVE) {
        return (void*) vdot8s_sve;
    }
#endif
#if defined(HWCAP_ASIMDDP) && defined(__ARM_NEON)
    if (hwcap & HWCAP_ASIMDDP) {
        return (void*) vdot8s_neon;
    }
#endif
    return (void*) vdot8s_scalar;
}

// dot8s will dispatch based on runtime capabilities
#if defined(__aarch64__) && defined(__linux__)
__attribute__((ifunc("resolve_dot8s")))
int32_t dotProduct(int8_t vec1[], int8_t vec2[], int32_t limit);
#else
```

RUNTIME RESOLUTION

Key Benefits

KEY BENEFITS

Zero runtime overhead

Resolution happens **once at load time** via IFUNC. Every subsequent call goes directly to the best implementation — no branching, no checks.

Single binary

One **libdotProduct.so** works across **Graviton 2, 3 and 4**. No per-platform builds, no conditional packaging.

Automatic optimization

No code changes needed for new hardware — unless a faster implementation or new intrinsics exist to take advantage of it.

Graceful degradation

Always has a working fallback. If SVE and NEON are unavailable, scalar kicks in automatically — no crashes, no configuration.

Clean Java integration

Called via **FFM downcall** from Java — **no JNI code** required. Cleaner API, safer memory model, first-class Java 22+ support.

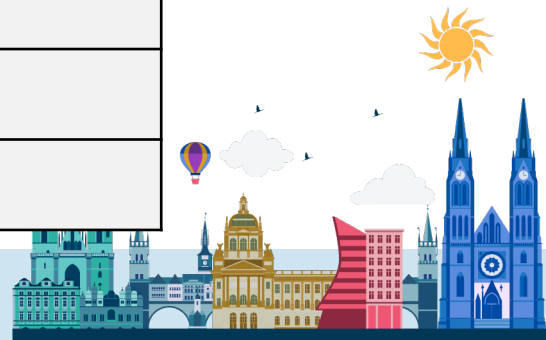
Performance Impact: JMH Benchmarks

Graviton 2

Dimension	Lucene Panama (Baseline) (ops/µs)	Native SIMD (Candidate) (ops/µs)	Speedup
256	7.7	48.3	6.3×
1024	2.2	23.6	10.8×
4096	0.6	7.6	**13.4×** ✨

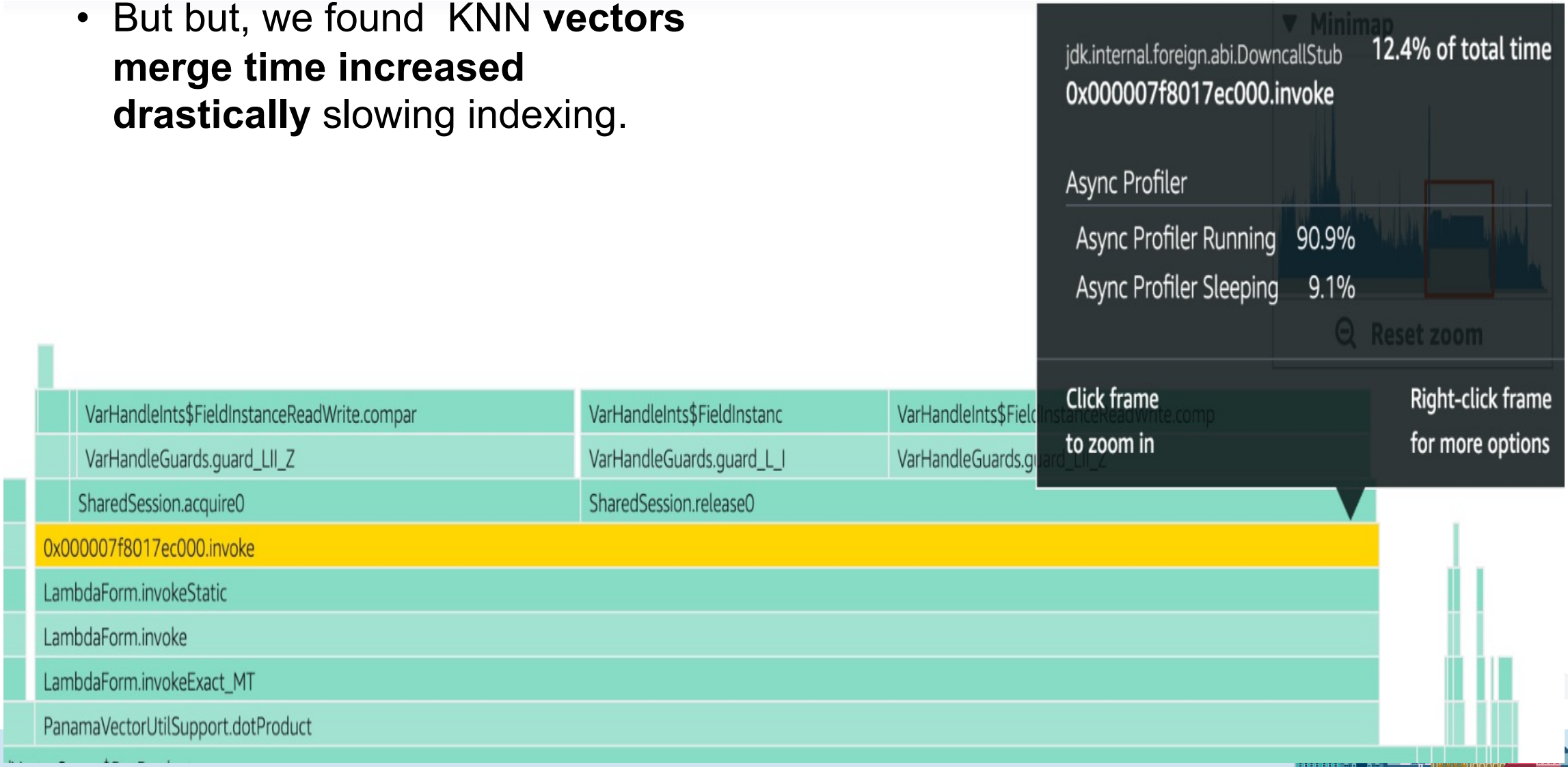
Graviton 3

Dimension	Lucene Panama (Baseline) (ops/µs)	Native SIMD (Candidate) (ops/µs)	Speedup
256	30.5	53.2	1.7×
1024	8.0	35.3	4.4×
4096	2.0	11.8	**5.8×** ✨



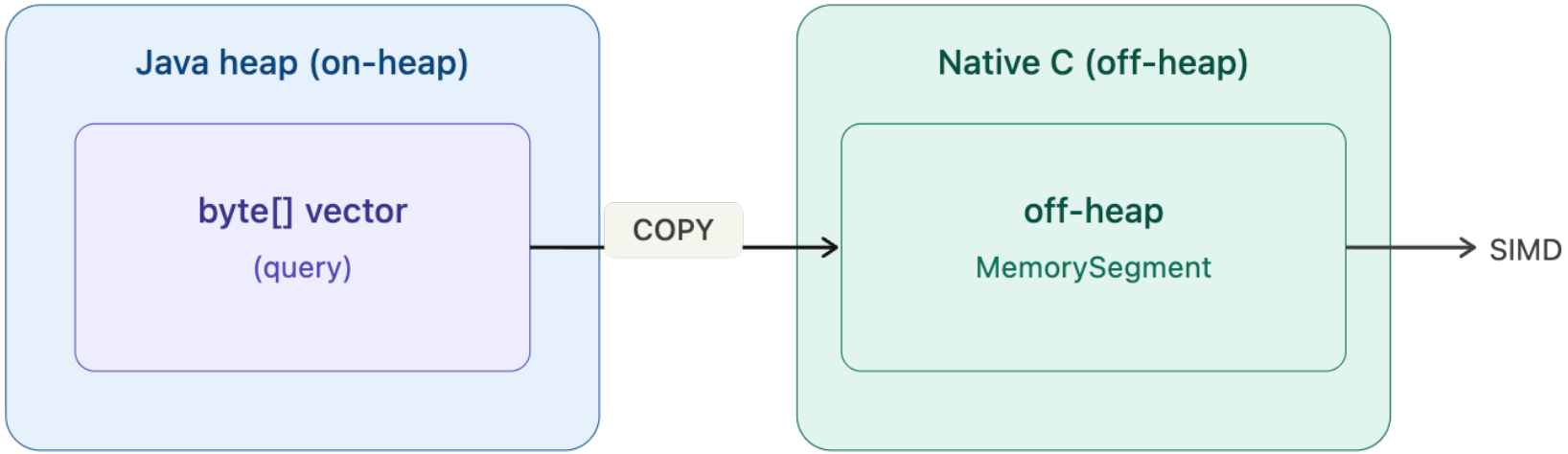
Memory copy overhead Issue

- But but, we found KNN **vectors merge time increased drastically** slowing indexing.



Memory copy overhead Issue

- **Root cause :** In <= Java 21 FFM API by default REJECTED on-heap MemorySegment's for native calls. It forces you to copy to off-heap first → the excessive memory copy overhead we hit.



⚠ FFM API requires off-heap memory for native calls

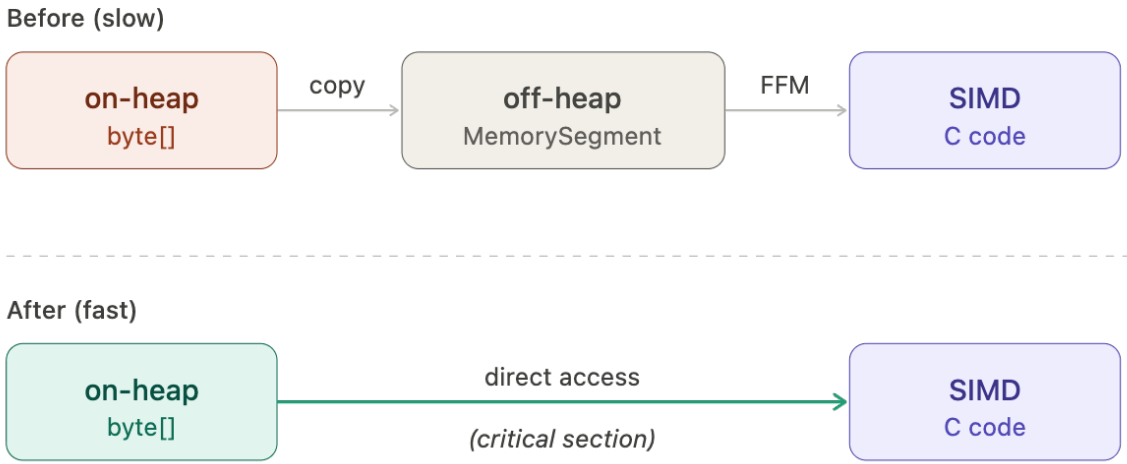
⚠ Extra copy on every dot product invocation

⚠ GC pressure from concurrent graph merges

Using Linker.Option.critical

In Java 22+, we could pass the on- heap MemorySegment's directly with native calls using ***Linker.Option.critical(allowHeapAccess = true)***

Feature	Without `critical`	With `critical`
Memory copies	1 per call	0 per call
GC pressure	High	None
GC paused	No	Yes (briefly)
Index-time perf	Degraded	Full SIMD gains



Using Linker.Option.critical

critical

```
static Linker.Option critical(boolean allowHeapAccess)
```

Returns a linker option used to mark a foreign function as *critical*.

A critical function is a function that has an extremely short running time in all cases (similar to calling an empty function), and does not call back into Java (e.g. using an upcall stub).

Using this linker option is a hint that some implementations may use to apply optimizations that are only valid for critical functions.

Using this linker option when linking non-critical functions is likely to have adverse effects, such as loss of performance or JVM crashes.

Critical functions can optionally allow access to the Java heap. This allows clients to pass heap memory segments as addresses, where normally only off-heap memory segments would be allowed. The memory region inside the Java heap is exposed through a temporary native address that is valid for the duration of the function call. Use of this mechanism is therefore only recommended when a function needs to do short-lived access to Java heap memory, and copying the relevant data to an off-heap memory segment would be prohibitive in terms of performance.

Parameters:

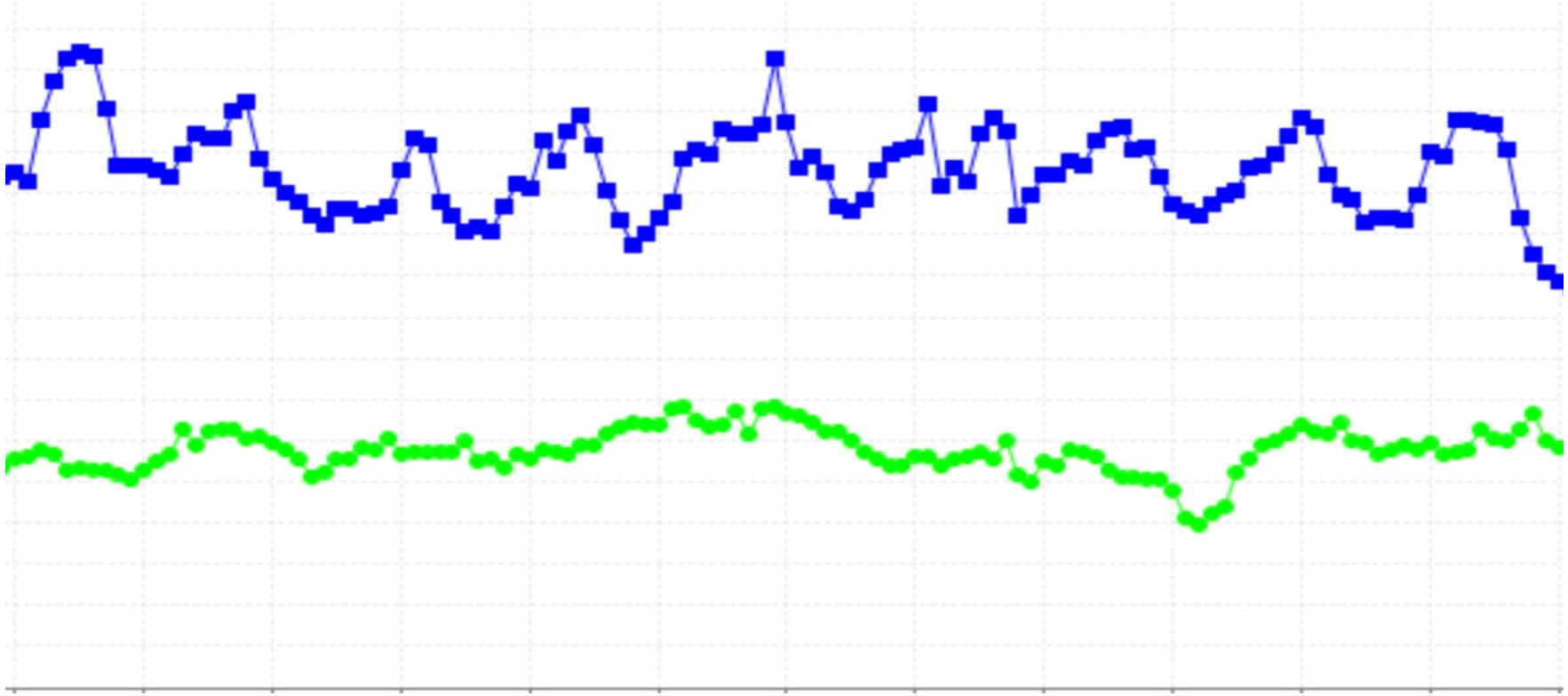
`allowHeapAccess` - whether the linked function should allow access to the Java heap.

Returns:

a linker option used to mark a foreign function as *critical*

Results from Prod (Search Latency)

- Average search latency improving by **60%**



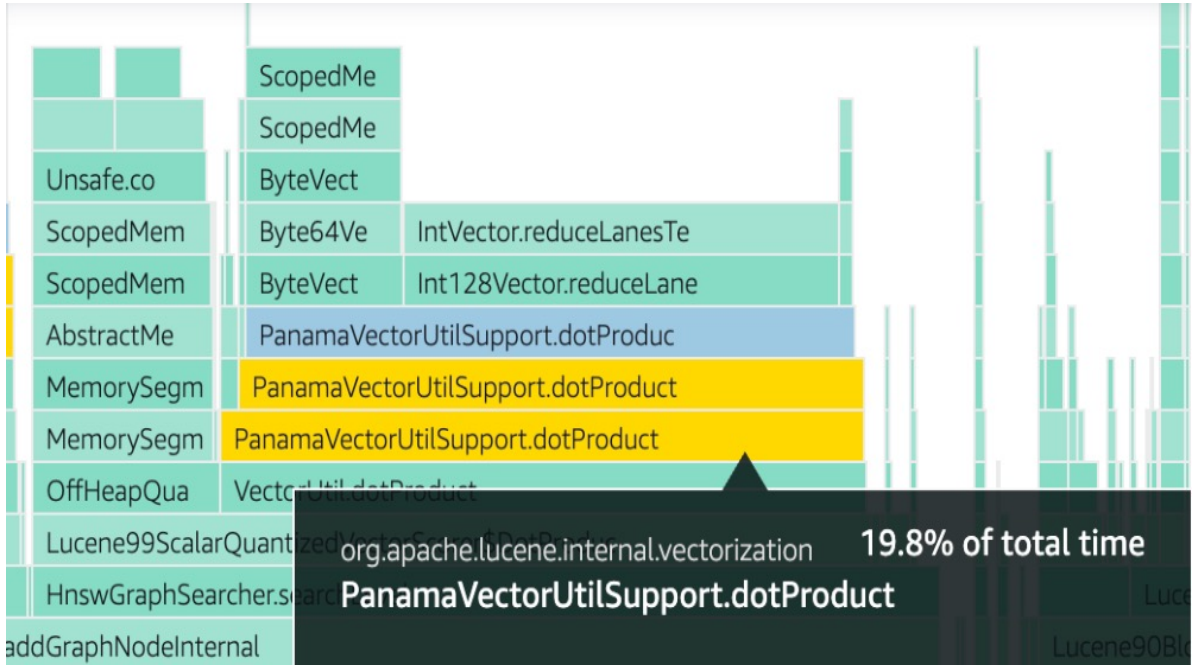
Baseline : Lucene Panama

Candidate : Native Vectorization

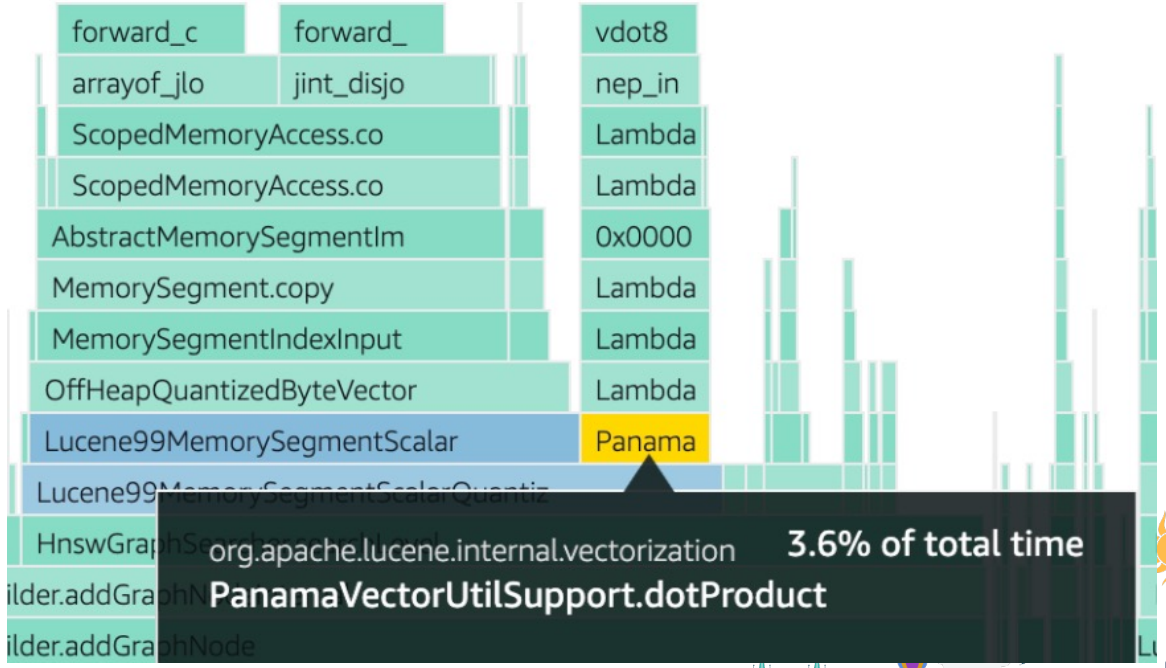
Results from Prod (CPU ticks)

- Profiler showed **5-6x** faster dot product computation times

- **Baseline**



- **Candidate**



JDK acknowledging lack of support for ARM

OpenSearchCon EUROPE

JDK issue : https://bugs.openjdk.org/browse/JDK-8377386

JDK / JDK-8377386

Vector: Support vector product instructions

Open

Export

Details

Type:	Enhancement	Resolution:	Unresolved
Priority:	P4	Fix Version/s:	tbd
Affects Version/s:	None		
Component/s:	hotspot		
Labels:	amazon-interest performance vectorapi		
Subcomponent:	compiler		

People

Assignee:	Unassigned
Reporter:	Aleksey Shipilev
Votes:	0 Vote for this issue
Watchers:	2 Start watching this issue

Dates

Created:	2026-02-06 08:43
Updated:	2026-03-20 16:31

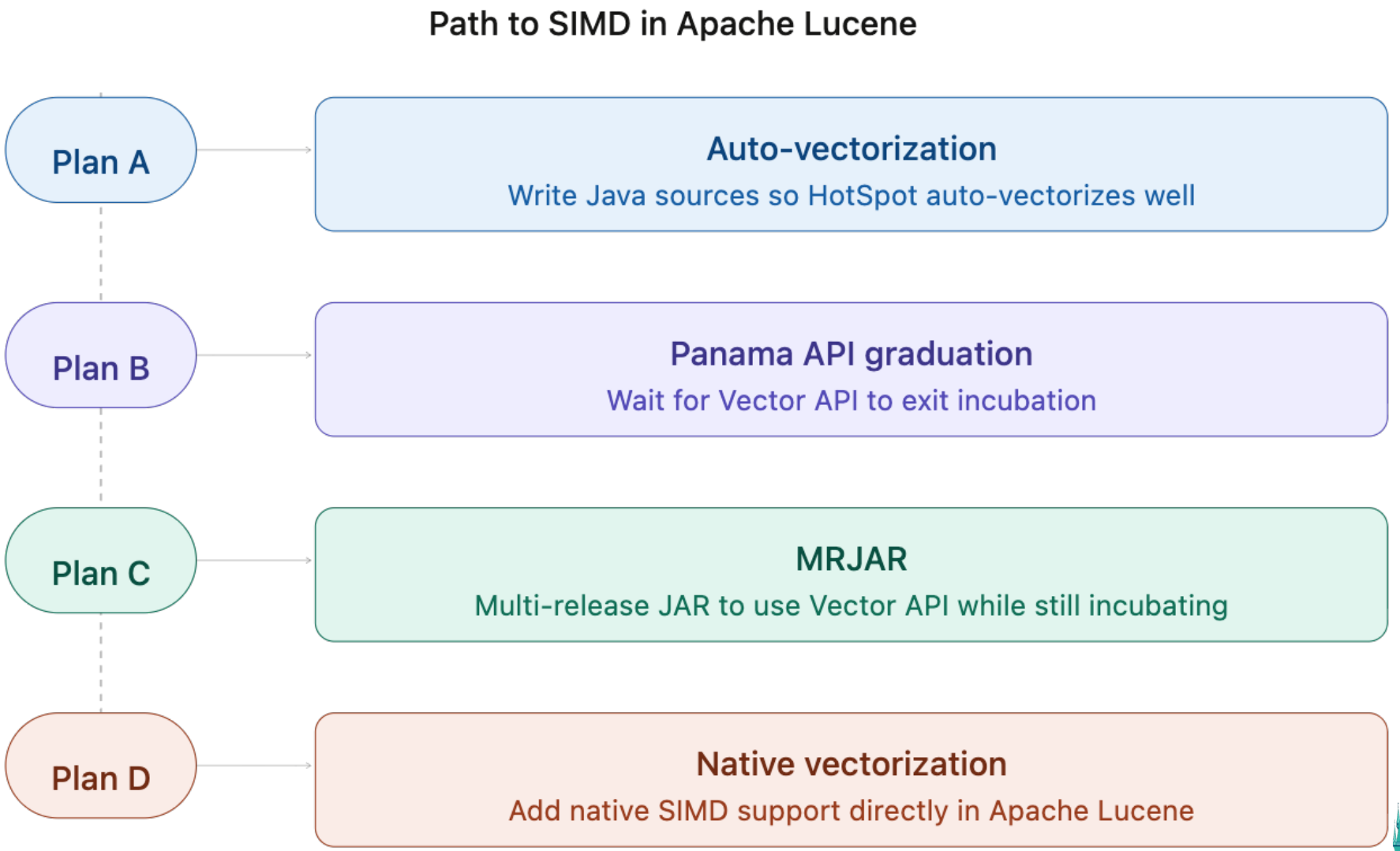
Description

Lucene folks ran into the absence of high-performance dot product support:
https://github.com/apache/lucene/pull/13572

They apparently end up calling into native implementations, where GCC is able to auto-vectorize to hardware instructions like AArch64 SDOT:
https://developer.arm.com/documentation/ddi0602/2025-12/SME-Instructions/SDOT--4-way--multiple-and-indexed-vector---Multi-vector-signed-integer-dot-product-by-indexed-element-

This might be a good addition to Vector APIs.

Apache Lucene's approach to SIMD



Lucene 11 : Native SIMD support in Apache Lucene

OpenSearchCon EUROPE

PR #15508 — apache/lucene

No C code in Lucene

Remains a pure Java project
No added build complexity

Decoupled binary

Native library is separate
Lucene works without it

No platform binaries shipped

Lucene defines the interface
Users bring their own .so
Open door for vendor experimentation

Zero-copy native calls

Uses Linker.Option.critical
allowHeapAccess = true
No on-heap → off-heap copy

In short

Lucene	"Here's a way to plug in native SIMD"
Users	"Here's my optimized binary for my HW"
Result	Best of both worlds — clean + fast

Lucene 11 : Native SIMD support in Apache Lucene

OpenSearchCon EUROPE

1. **Compile** the C code (default impl or custom)

```c
int32_t dot8s(int8_t vec1[], int8_t vec2[], int32_t limit) {
    // Scalar fallback
    int32_t result = 0;
    for (int32_t i = 0; i < limit; i++) {
        result += vec1[i] * vec2[i];
    }
    return result;
}
```

```
// Linux/Unix
gcc -shared -O3 -march=native -funroll-loops -o libdotProduct.so dotProduct.c

// macOS
gcc -shared -O3 -march=native -funroll-loops -o libdotProduct.dylib dotProduct.c

// Windows
gcc -shared -O3 -march=native -funroll-loops -o dotProduct.dll dotProduct.c
```

2. **Test** if everything works and dot product implementation

```
./gradlew test                                  // PASSES, default native dot product is off
./gradlew test -Ptest.native.dotProduct=false   // PASSES, switched off
./gradlew test \
  -Ptest.native.dotProduct=true \
  -Ptests.jvmargs="-Djava.library.path=/home/simd" // PASSES


./gradlew test -Ptest.native.dotProduct=true    // FAILS, needs library to be linked
```

3. **Benchmark** against the other C implementation or Panama

```
java --enable-native-access=ALL-UNNAMED \
  --enable-preview \
  -Djava.library.path="/home/simd" \
  -jar lucene/benchmark-jmh/build/benchmarks/lucene-benchmark-jmh-11.0.0-SNAPSHOT.jar \
  regexp "binaryDotProductVector|dot8sNative"
```

Benefits to OpenSearch

Benefits to OpenSearch

Less maintenance

Drop custom C code & scorers — use Lucene's native interface. No more parallel SIMD implementation to maintain.

Automatic upstream improvements

Lucene community optimizes → OpenSearch benefits for free. Bug fixes, new intrinsics, new platform support.

Single source of truth

One native interface instead of diverging codebases. Reduces risk of subtle scoring inconsistencies.

Easier upgrades

Lucene version bumps no longer risk breaking the custom native layer. Smoother adoption of new releases.

Community-driven testing

Broader adoption = more eyes, more hardware coverage. Better correctness and performance validation across platforms.

OpenSearchCon EUROPE

THANK YOU!