- **Runtime systems:** The runtime system needs to be able to manage and schedule computations on different types of hardware. This involves developing runtime systems that can handle the complexities of heterogeneous hardware, such as managing memory across different types of devices and scheduling computations to minimize data transfer. Consider the task of image processing, in which different filters or transformations are applied to an image. The way these operations are scheduled on a GPU can drastically affect performance. For instance, processing an image in smaller segments or "blocks" might be more efficient for certain filters, whereas other operations might benefit from processing the image as larger contiguous chunks. The choice of how to divide and process the image—whether in smaller blocks or in larger sections—can be analogous to the difference between a filter being applied in real time versus experiencing a noticeable delay. In the runtime system, such computations must be adeptly scheduled to minimize data transfer overheads and to fully harness the hardware's capabilities.
- **Interoperability:** The language and toolchain should provide mechanisms that allow for interoperability with the existing libraries and frameworks that are designed to work with exotic hardware. This can involve providing Foreign Function Interface (FFI) mechanisms, like those in Project Panama, that allow Java code to interoperate with native libraries.
- **Library support:** Although some libraries have been developed to support exotic hardware, these libraries are often specific to certain types of hardware and may not be available or optimized for all types of hardware offered by cloud providers. Specializations in these libraries can include optimizations for specific hardware architectures, which can lead to significant performance differences when running on different types of hardware.

These considerations significantly influence the design and implementation of the projects that aim to better leverage exotic hardware capabilities. For instance, the design of Aparapi, an API for expressing data parallel workloads, has been heavily influenced by the need for language abstractions that can express parallelism and take advantage of the unique features of exotic hardware. Similarly, the development of TornadoVM has been guided by the need for a runtime system that can manage and schedule computations on different types of hardware.

In the case studies that follow, we will delve into these projects in more detail, exploring how they address the challenges of exotic hardware utilization and how they are influenced by the considerations discussed earlier.

Case Studies

It's important to ground our discussion of exotic hardware and the JVM in real-world examples, and what better way to do that than through a series of case studies. Each of these projects—LWJGL, Aparapi, Project Sumatra, CUDA4J (the joint IBM–NVIDIA project), TornadoVM, and Project Panama—offers unique insights into the challenges and opportunities presented by hardware accelerators.

- **LWJGL (Lightweight Java Game Library)**[12] serves as a foundational case, demonstrating the use of the Java Native Interface (JNI) to enable Java applications to interact with

[12] www.lwjgl.org

native APIs for a range of tasks, including graphics and compute operations. It provides a practical example of how existing JVM mechanisms can be used to leverage specialized hardware.

- **Aparapi** showcases how language abstractions can be designed to express parallelism and take advantage of the unique features of heterogeneous hardware. It translates Java bytecode into OpenCL at runtime, enabling the execution of parallel operations on the GPU.
- **Project Sumatra**, although no longer active, was a significant effort to enhance the JVM's ability to offload computations to the GPU by leaning on Java 8's Stream API to express parallelism. The experiences and lessons from Project Sumatra continue to inform current and future projects.
- Running in parallel to Project Sumatra, **CUDA4J** emerged as a joint effort by IBM and NVIDIA.[13] This project leveraged the Java 8 Stream API, enabling Java developers to write GPU computations that the CUDA4J framework translated into CUDA kernels. It demonstrated the power of collaborative efforts in the community to enhance the compatibility between the JVM and exotic hardware, particularly GPUs.
- **TornadoVM** demonstrates how runtime systems can be developed to manage and schedule computations on different types of hardware. It provides a practical solution to the challenges of hardware heterogeneity and API compatibility.
- **Project Panama** provides a glimpse into the future of the JVM. It focuses on improving the connection between the JVM and foreign APIs, including libraries and hardware accelerators, through the introduction of a new FFM API and a Vector API. These developments represent a significant evolution in JVM design, enabling more efficient and streamlined interactions with exotic hardware.

These projects were chosen not only for their technical innovations but also for their relevance to the evolving landscape of JVM and specialized hardware. They represent significant efforts in the community to adapt the JVM to better leverage the capabilities of exotic hardware, and as such, provide valuable insights for our discussion.

LWJGL: A Baseline Example

LWJGL serves as a prime example of how Java interacts with exotic hardware. This mature and widely used library provides Java developers with access to a range of native APIs, including those for graphics (OpenGL, Vulkan), audio (OpenAL), and parallel computation (OpenCL). Developers can use LWJGL in their Java applications to access these native APIs. For instance, a developer might use LWJGL's bindings for OpenGL to render 3D graphics in a game or simulation. This involves writing Java code that calls methods in the LWJGL library, which in turn invoke the corresponding functions in the native OpenGL library.

[13] Despite its niche nature, CUDA4J continues to find application on IBM Power platforms with NVIDIA hardware: www.ibm.com/docs/en/sdk-java-technology/8?topic=only-cuda4j-application-programming-interface-linux-windows.

Here is a simple example of how one might use LWJGL to create an OpenGL context and clear the screen:

```
try (MemoryStack stack = MemoryStack.stackPush()) {
    GLFWErrorCallback.createPrint(System.err).set();
    if (!glfwInit()) {
        throw new IllegalStateException("Unable to initialize GLFW");
    }
    long window = glfwCreateWindow(800, 600, "Shrek: The Musical", NULL, NULL);
    glfwMakeContextCurrent(window);
    GL.createCapabilities();
    while (!glfwWindowShouldClose(window)) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // Set the color to swamp green
        glClearColor(0.13f, 0.54f, 0.13f, 0.0f);
        // Draw a simple 3D scene...
        drawScene();
        // Draw Shrek's house
        drawShrekHouse();
        // Draw Lord Farquaad's castle
        drawFarquaadCastle();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
}
```

In this example, we're creating a simple 3D scene for a game based on *Shrek: The Musical*. We start by initializing GLFW and creating a window. We then make the window's context current and create an OpenGL context. In the main loop of our game, we clear the screen, set the color to swamp green (to represent Shrek's swamp), draw our 3D scene, and then draw Shrek's house and Lord Farquaad's castle. We then swap buffers and poll for events. This is a very basic example, but it gives you a sense of how you might use LWJGL to create a 3D game in Java.

LWJGL and the JVM

LWJGL interacts with the JVM primarily through JNI, the standard mechanism for calling native code from Java. When a Java application calls a method in LWJGL, the LWJGL library uses JNI to invoke the corresponding function in the native library (such as `OpenGL.dll` or `OpenAL.dll`). This allows the Java application to leverage the capabilities of the native library, even though the JVM itself does not directly support these capabilities.

Figure 9.1 illustrates the flow of control and data between Java code, LWJGL, and the native libraries. At the top is the application that runs on top of the JVM. This application uses the Java APIs provided by LWJGL to interact with the native APIs. The LWJGL provides a bridge between the JVM and the native APIs, allowing the application to leverage the capabilities of the native hardware.

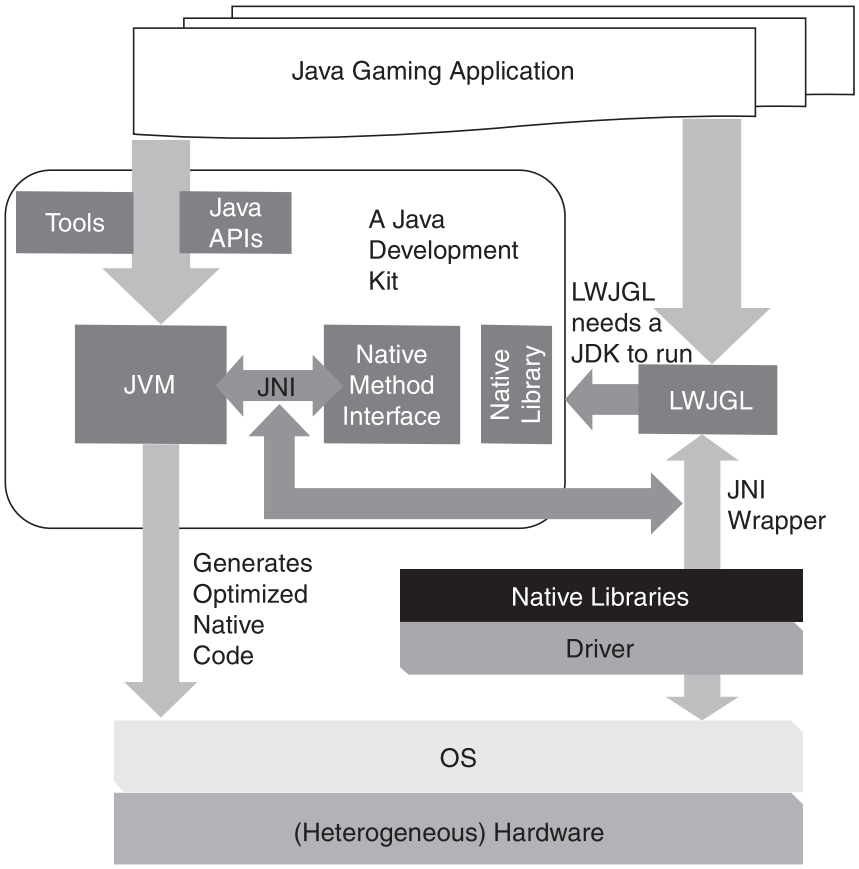

Figure 9.1 LWJGL and the JVM

The LWJGL uses JNI to call native libraries—represented by the JNI Wrapper in Figure 9.1. The JNI Wrapper is essentially "glue code" that converts variables between Java and C and calls the native libraries. The native libraries, in turn, interact with the drivers that control the hardware. In the case of LWJGL, this interaction encompasses everything from rendering visuals and processing audio to supporting various parallel computing tasks.

The use of JNI has both benefits and drawbacks. On the positive side, JNI allows Java applications to access a wide range of native libraries and APIs, enabling them to leverage exotic hardware capabilities that are not directly supported by the JVM. However, JNI also introduces overhead, as calls to native methods are generally slower than calls to Java methods. Additionally, JNI requires developers to write and maintain glue code that bridges the gap between Java and native code, which can be complex and error prone.

Challenges and Limitations

Although JNI has been successful in providing Java developers with access to native APIs, it also highlights some of the challenges and limitations of this approach. One key challenge is the complexity of working with native APIs, which often have different design philosophies and conventions than Java. This can lead to inefficiencies, especially when dealing with low-level details such as memory management and error handling.

Additionally, LWJGL's reliance on JNI to interface with native libraries comes with its own set of challenges. These challenges are highlighted by the insights of Gary Frost, an architect at Oracle, who brings a rich history in bridging Java with exotic hardware:

- **Memory management mismatch:** The JVM is highly autonomous in managing memory. It controls the memory allocation and deallocation life cycle, providing automatic garbage collection to manage object lifetimes. This control, however, can clash with the behavior of native libraries and hardware APIs that require manual control over memory. APIs for GPUs and other accelerators often rely on asynchronous operations, which can lead to a mismatch with the JVM's memory management style. These APIs commonly allow data movement requests and kernel dispatches to return immediately, with the actual operations being queued for later execution. This latency-hiding mechanism is crucial for achieving peak performance on GPUs and similar hardware, but its asynchronous nature conflicts with the JVM's garbage collector, which may relocate Java objects at any time. The pointers passed to the GPU through JNI may become invalid in mid-operation, necessitating the use of functions like `GetPrimitiveArrayCritical` to "pin" the objects and prevent them from being moved. However, this function can pin objects only until the corresponding JNI call returns, forcing Java applications to artificially wait until data transfers are complete, thereby hindering the performance benefits of asynchronous operations. It's worth noting that this memory management mismatch is not unique to LWJGL. Other Java frameworks and tools, such as Aparapi, and TornadoVM (as of this writing), also face similar challenges in this regard.
- **Performance overhead:** Transitions between Java and native code via JNI are generally slower than pure Java calls due to the need to convert data types and manage different calling conventions. This overhead can be relatively minor for applications that make infrequent calls to native methods. However, for performance-critical tasks that rely heavily on native APIs, JNI overhead can become a substantial performance bottleneck. The forced synchronization between the JVM and native libraries merely exacerbates this issue.
- **Complexity and maintenance:** Using JNI requires a firm grasp of both Java and C/C++ because developers must write "glue code" to bridge the two languages. This need for bilingual proficiency and the translation between differing paradigms can introduce complexities, making writing, debugging, and maintaining JNI code a challenging task.

LWJGL and JNI remain instrumental in allowing Java applications to access native libraries and leverage exotic hardware capabilities, but understanding their nuances is crucial, especially in performance-critical contexts. Careful design and a thorough understanding of both the JVM and the target native libraries can help developers navigate these challenges and harness the full power of exotic hardware through Java. LWJGL provides a baseline against which we can compare other approaches, such as Aparapi, Project Sumatra, and TornadoVM, which we will discuss in the following sections.

Aparapi: Bridging Java and OpenCL

Aparapi, an acronym for "A PARallel API," is a Java API that allows developers to implement parallel computing tasks. It serves as a bridge between Java and OpenCL. Aparapi provides a way

for Java developers to offload computation to GPUs or other OpenCL-capable devices, thereby leveraging the remarkable coordinated computation power of these devices.

Using Aparapi in Java applications involves defining parallel tasks using annotations and classes provided by the Aparapi API. Once these tasks are defined, Aparapi takes care of translating the Java bytecode into OpenCL, which can then be executed on the GPU or other hardware accelerators.

To truly appreciate Aparapi's capabilities, let's venture into the world of astronomy, where an adaptive optics technique is used in telescopes to enhance the performance of optical systems by adeptly compensating for distortions caused by wavefront aberrations.[14] A pivotal element of an adaptive optics system is the deformable mirror, a sophisticated device that is used to correct the distorted wavefronts.

In my work at the Center for Astronomical Adaptive Optics (CAAO), we used adaptive optics systems to correct for atmospheric distortions in real time. This task required processing large amounts of sensor data to adjust the shape of the telescope's mirror—a task that could greatly benefit from parallel computation. Aparapi could potentially be used to offload these computations to a GPU or other OpenCL-capable device, enabling the CAAO team to process the wavefront data in parallel. This would significantly speed up the correction process and allow the team to adjust the telescope more quickly and accurately to changing atmospheric conditions.

```
import com.amd.aparapi.Kernel;
import com.amd.aparapi.Range;

public class WavefrontCorrection {
    public static void main(String[] args) {
        // Assume wavefrontData is a 2D array representing the distorted wavefront
        final float[][] wavefrontData = getWavefrontData();

        // Create a new Aparapi Kernel, a unit of computation designed for wavefront correction
        Kernel kernel = new Kernel() {
          // The run method defines the computation. It will be executed in parallel on the GPU.
            @Override
            public void run() {
               // Get the global id, a unique identifier for each work item (computation)
                int x = getGlobalId(0);
                int y = getGlobalId(1);

                // Perform the wavefront correction computation.
                // This is a placeholder; the actual computation would depend on the specifics of
                   the adaptive optics system
                wavefrontData[x][y] = wavefrontData[x][y] * 2;
                // In this example, each pixel of the wavefront data is simply doubled.
```

[14] https://en.wikipedia.org/wiki/Adaptive_optics

```
                // In real-world applications, implement your wavefront correction algorithm here.
            }
        };

        // Execute the kernel with a Range representing the size of the wavefront data
        // Range.create2D creates a two-dimensional range equal to the size of the wavefront data.
        // This determines the work items (computations) that will be parallelized for the GPU.
        kernel.execute(Range.create2D(wavefrontData.length, wavefrontData[0].length));
    }
}
```

In this code, `getWavefrontData()` is a method that retrieves the current wavefront data from the adaptive optics system. The computation inside the `run()` method of the kernel would be replaced with the actual computation required to correct the wavefront distortions.

NOTE When working with Aparapi, it's essential for Java developers to have an understanding of the OpenCL programming and execution models. This is because Aparapi translates the Java code into OpenCL to run on the GPU. The code snippet shown here demonstrates how a wavefront correction algorithm can be implemented using Aparapi, but numerous intricacies must also be considered in such a use case. For instance, 2D Java arrays might not be directly supported in Aparapi due to the need to flatten memory for GPU execution. Additionally, this code won't execute in a standard sequential Java manner; instead, a wrapper method is required to invoke this code if the function is unoptimized. This highlights the importance of being aware of the underlying hardware and execution models when leveraging Aparapi for GPU acceleration.

Aparapi and the JVM

Aparapi offers a seamless integration with the JVM by converting Java bytecode into executable OpenCL code that can be executed on the GPU or other hardware accelerators. This translation is performed by the Aparapi compiler and runtime, which are optimized for OpenCL. The JVM then offloads the execution of this code to the GPU via Aparapi and OpenCL, enabling Java applications to leverage significant performance improvements for data parallel workloads.

Figure 9.2 illustrates the flow from a Java application using Aparapi to computation offloading. The left side of the diagram shows the JVM, within which our Java application operates. The application (on the right) utilizes Aparapi to define a *Kernel*[15] that performs the computation we want to offload. The "OpenCL-Enabled Compiler and Runtime" then translates this *Kernel* from Java bytecode into OpenCL code. The resultant OpenCL code is executed on the GPU or another OpenCL-capable device, as depicted by the OpenCL arrow leading to the GPU.

[15] https://www.javadoc.io/doc/com.aparapi/aparapi/1.9.0/com/aparapi/Kernel.html

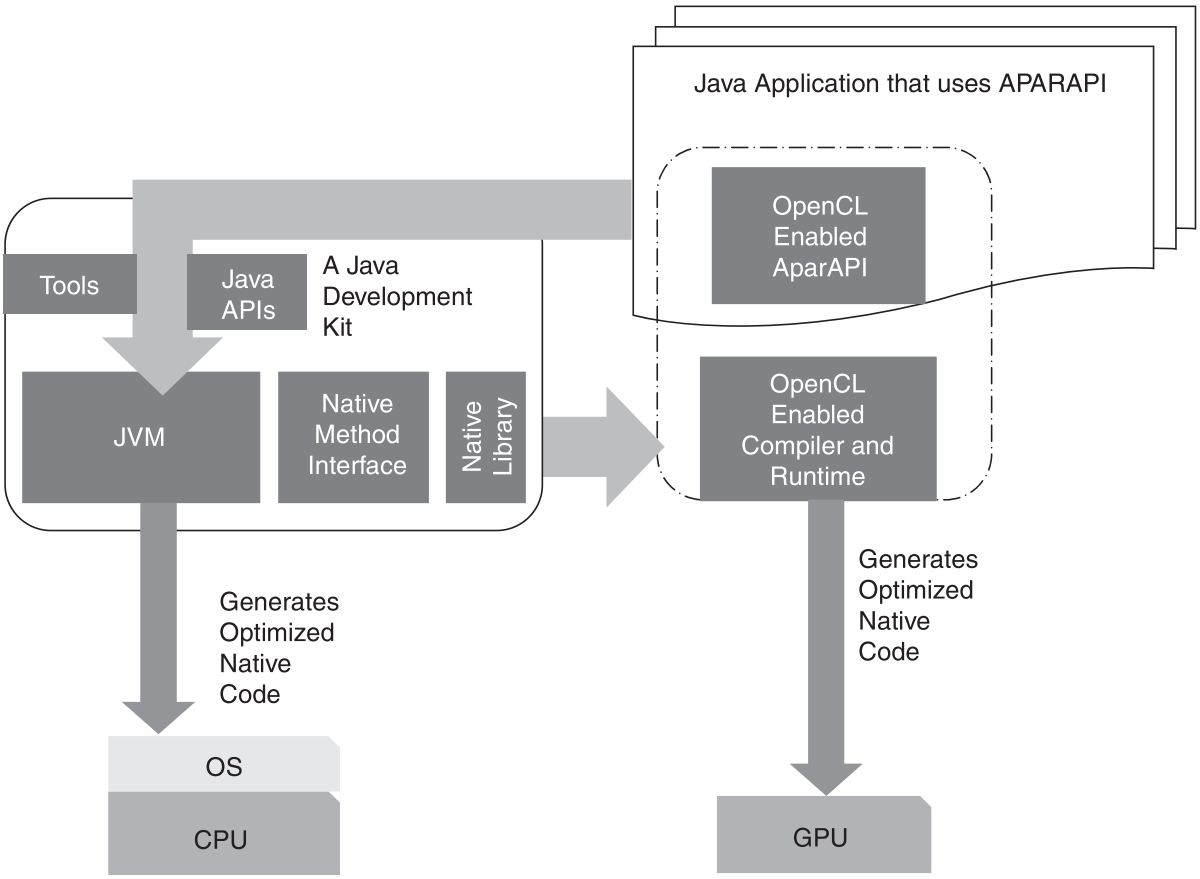

Figure 9.2 Aparapi and the JVM

Challenges and Limitations

Aparapi offers a promising avenue for Java developers to harness the computational prowess of GPUs and other hardware accelerators. However, like any technology, it presents its own set of challenges and limitations:

- **Data transfer bottleneck:** One of the main challenges lies in managing the data transfer between the CPU and the GPU. Data transfer can be a significant bottleneck, especially when working with large amounts of data. Aparapi provides mechanisms such as the ability to specify array access types (read-only, write-only, or read-write on the GPU) to help optimize the data transfer.
- **Explicit memory management:** Java developers are accustomed to automatic memory management courtesy of the garbage collector. In contrast, OpenCL demands explicit memory management, introducing a layer of complexity and potential error sources. Aparapi's approach, which translates Java bytecode into OpenCL, does not support dynamic memory allocation, further complicating this aspect.

- **Memory limitations:** Aparapi cannot exploit the advantages of constant or local memory, which can be crucial for optimizing GPU performance. In contrast, TornadoVM automatically leverages these memory types through JIT compiler optimizations.[16]
- **Java subset and anti-pattern:** Aparapi supports only a subset of the Java language. Features such as exceptions and dynamic method invocation are not supported, requiring developers to potentially rewrite or refactor their code. Furthermore, the Java kernel code, while needing to pass through the Java compiler, is not actually intended to run on the JVM. As Gary Frost puts it, this approach of using Java syntax to represent "intent" but not expecting the JVM to execute the resulting bytecode represents an anti-pattern in Java and can make it difficult to leverage GPU features effectively.

It's important to recognize that these challenges are not unique to Aparapi. Platforms such as TornadoVM, OpenCL, CUDA, and Intel oneAPI[17] also support specific subsets of their respective languages, often based on C/C++. In consequence, developers working with these platforms need to be aware of the specific features and constructs that are supported and adjust their code accordingly.

In conclusion, Aparapi represents a step forward in enabling Java applications to leverage the power of exotic hardware. Ongoing development and improvements are likely to address current challenges, making it even easier for developers to harness the power of GPUs and other hardware accelerators in their Java applications.

Project Sumatra: A Significant Effort

Project Sumatra was a pioneering initiative in the landscape of JVM and high-performance hardware. Its primary goal was to enhance the JVM's ability to work with GPUs and other accelerators. The project aimed to enable Java applications to offload data parallel tasks to the GPU directly from within the JVM itself. This was a significant departure from the traditional model of JVM execution, which primarily targeted CPUs.

Project Sumatra introduced several key concepts and components to achieve its goals. One of the most significant was the Heterogeneous System Architecture (HSA) and the HSA Intermediate Language (HSAIL). HSAIL is a portable intermediate language that is finalized to hardware ISA at runtime. This enabled the dynamic generation of optimized native code for GPUs and other accelerators.

Furthermore, the coherency between CPU and GPU caches provided by HSA obviated the need for moving data across the system bus to the accelerator, streamlining the offloading process for Java applications and enhancing performance.

Another key component of Project Sumatra was the Graal JIT compiler, which we briefly explored in Chapter 8, "Accelerating Time to Steady State with OpenJDK HotSpot VM." Graal

[16] Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. *Automatically Exploiting the Memory Hierarchy of GPUs Through Just-in-Time Compilation*. Paper presented at the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21), April 16, 2021. https://research.manchester.ac.uk/files/190177400/MPAPADIMITRIOU_VEE2021_GPU_MEMORY_JIT_Preprint.pdf.

[17] www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

is a dynamic compiler that can be used as a JIT compiler within the JVM. Within Sumatra's framework, Graal was used to generate HSAIL code from Java bytecode, which could then be executed on the GPU.

Project Sumatra and the JVM

Project Sumatra was designed to be integrated closely with the JVM. It explored the integration of components such as the "Graal JIT Backend" and "HSA Runtime" into the JVM architecture, as depicted in Figure 9.3. These enhancements to the architecture allowed Java bytecode to be compiled into HSAIL code, which the HSA runtime could use to generate optimized native code that could run on the GPU (in addition to the existing JVM components that generate optimized native code for the CPU).

A prime target of Project Sumatra was the enhancement of the Java 8 Stream API to execute on the GPU, where the operations expressed in the Stream API would be offloaded to the GPU for processing. This approach would be particularly beneficial for compute-intensive tasks such as correcting numerous wavefronts in the astronomy scenario described earlier—the parallelized computation capabilities of the GPU could be leveraged to perform these corrections more quickly than would be possible on the CPU. However, to utilize these enhancements, specialized hardware that supports the HSA infrastructure, like AMD's Accelerated Processing Units (APUs), was required.

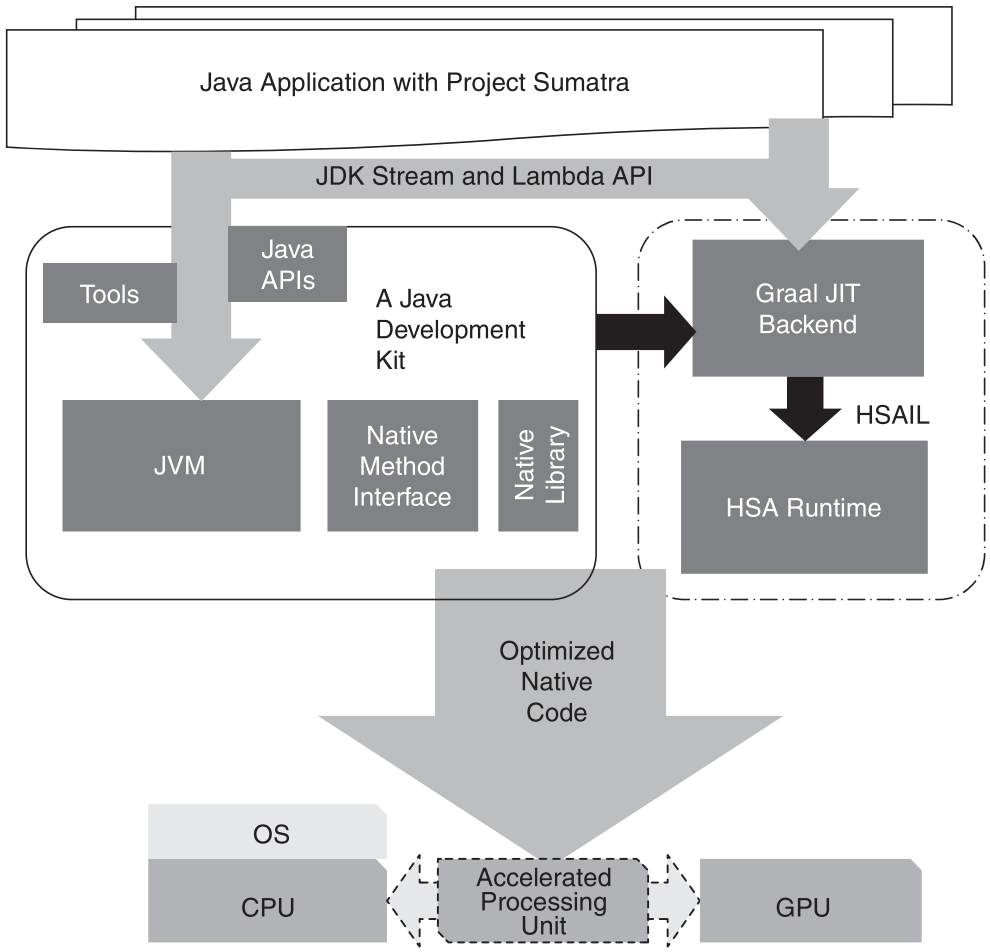

Figure 9.3 Project Sumatra and the JVM

In the following example, we use the Stream API to create a parallel stream from a list of wavefronts. Then we use a lambda expression, `wavefront -> wavefront.correct()`, to correct each wavefront. The corrected wavefronts are collected into a new list.

```
// Assume we have a List of Wavefront objects called wavefronts
List<Wavefront> wavefronts = ...;

// We can use the Stream API to process these wavefronts in parallel
List<Wavefront> correctedWavefronts = wavefronts.parallelStream()
    .map(wavefront -> {
        // Here, we're using a lambda expression to define the correction operation
        Wavefront correctedWavefront = wavefront.correct();
        return correctedWavefront;
    })
    .collect(Collectors.toList());
```

In a traditional JVM, this code would be executed in parallel on the CPU. However, with Project Sumatra, the goal was to allow such data parallel computations to be offloaded to the GPU. The JVM would recognize that this computation could be offloaded to the GPU, generate the appropriate HSAIL code, and execute it on the GPU.

Challenges and Lessons Learned

Despite its ambitious goals, Project Sumatra faced several challenges. One of the main challenges was the complexity of mapping Java's memory model and exception semantics to the GPU. Additionally, the project was closely tied to the HSA, which limited its applicability to HSA-compatible hardware.

A significant part of this project was led by Tom Deneau, Eric Caspole, and Gary Frost. Tom, with whom I had the privilege of working at AMD along with Gary, was the team lead for Project Sumatra. Beyond being just a colleague, Tom was also a mentor, whose deep understanding of the Java memory model and guidance played an instrumental role in Project Sumatra's accomplishments. Collectively, Tom, Eric, and Gary's extraordinary efforts allowed Project Sumatra to push the boundaries of Java's integration with the GPU, implementing advanced features such as thread-local allocation on the GPU and safe pointing from the GPU back to the interpreter.

However, despite the significant progress made, Project Sumatra was ultimately discontinued. The complexity of modifying the JVM to support GPU execution, along with the challenges of keeping pace with rapidly evolving hardware and software ecosystems, led to this decision.

In the realm of JVM and GPU interaction, the J9/CUDA4J offering deserves a noteworthy mention. The team behind J9/CUDA4J decided to use a Stream-based programming model similar to that included in Project Sumatra, and they continue to support their solution to date. Although J9/CUDA4J extends beyond the scope of this book, acknowledging its existence and contribution paints a more comprehensive picture of Java's journey toward embracing GPU computation.

Despite the discontinuation of Project Sumatra, its contributions to the field remain invaluable. It demonstrated the potential benefits of offloading computation to the GPU, while shedding light on the challenges that must be addressed to fully harness these benefits. The knowledge gained from Project Sumatra continues to inform ongoing and future projects in the area of JVM and hardware accelerators. As Gary puts it, we pushed hard on the JVM through Project Sumatra, and you can see its "ripples" in projects like the Vector API, Value Types, and Project Panama.

TornadoVM: A Specialized JVM for Hardware Accelerators

TornadoVM is a plug-in to OpenJDK and GraalVM that allows developers to run Java programs on heterogeneous or specialized hardware. According to Dr. Fumero, TornadoVM's vision actually extends beyond mere compilation for exotic hardware: It aims to achieve both code and performance portability. This is realized by harnessing the unique features of accelerators, encompassing not just the compilation process but also intricate aspects like data management and thread scheduling. By doing so, TornadoVM seeks to provide a holistic solution for Java developers delving into the realm of heterogeneous hardware computing.

TornadoVM offers a refined API that allows developers to express parallelism in their Java applications. This API is structured around tasks and annotations that facilitate parallel execution. The `TaskGraph`, which is included in the newer versions of TornadoVM, is central to defining the computation, while the `TornadoExecutionPlan` specifies execution parameters. The `@Parallel` annotation can be used to mark methods that should be offloaded to accelerators.

Revisiting our adaptive optics system scenario for a large telescope, suppose we have a large array of wavefront sensor data that requires real-time processing to correct for atmospheric distortions. Here's an updated example of how you might use TornadoVM in this context:

```
import uk.ac.manchester.tornado.api.TaskGraph;

public class TornadoExample {
    public static void main(String[] args) {
        // Create a task graph
        TaskGraph taskGraph = new TaskGraph("s0")
           .transferToDevice(DataTransferMode.FIRST_EXECUTION, wavefronts)
           .task("t0", TornadoExample::correctWavefront, wavefronts, correctedWavefronts)
           .transferToHost(DataTransferMode.EVERY_EXECUTION, correctedWavefronts);

        // Execute the task graph
        ImmutableTaskGraph itg = taskGraph.snapshot();
        TornadoExecutionPlan executionPlan = new TornadoExecutionPlan(itg);

        executionPlan.execute();
    }
```

```
    public static void correctWavefront(float[] wavefronts, float[] correctedWavefronts, int N) {
        for (@Parallel int i = 0; i < wavefronts.length; i++) {
           for (@Parallel int j = 0; j < wavefronts[i].length; j++) {
               correctedWavefronts[i * N + j] = wavefronts[i * N + j] * 2;
           }
        }
    }
}
```

In this example, a `TaskGraph` named `s0` is created; it outlines the sequence of operations and data transfers. The `wavefronts` data is transferred to the device (like a GPU) before computation starts. The `correctWavefront` method, added as a task to the graph, processes each wavefront to correct it. Post computation, the corrected data is transferred back to the host (like a CPU).

According to Dr. Fumero, TornadoVM (like Aparapi) doesn't support user-defined objects, but rather uses a collection of predefined ones. In our example, `wavefronts` would likely be a `float` array. Java developers using TornadoVM should be acquainted with OpenCL's programming and execution models. The code isn't executed in typical sequential Java fashion. If the function is deoptimized, a wrapper method is needed to invoke this code. A challenge arises when handling 2D Java arrays: The memory might require flattening unless the GPU can re-create the memory layout—a concern reminiscent of the issues with Aparapi.

TornadoVM and the JVM

TornadoVM is designed to be integrated closely with the JVM. It extends the JVM to exploit the expansive concurrent computation power of specialized hardware, thereby boosting the performance of JVM-based applications.

Figure 9.4 offers a comprehensive summary of the architecture of TornadoVM at a very high level. As you can see this diagram, TornadoVM introduces several new components to the JVM, each designed to optimize Java applications for execution on heterogenous hardware:

- **API (task graphs, execution plans, and annotations):** Developers use this interface to define tasks that can be offloaded to the GPU. Tasks are defined using Java methods, which are annotated to indicate that they can be offloaded, as shown in the earlier example.
- **Runtime (data optimizer + bytecode generation):** The TornadoVM runtime is responsible for optimizing data movement between the host (CPU) and the device (GPU). It uses a task-based programming model in which each task is associated with a data descriptor. The data descriptor provides information about the data size, shape, and type, which TornadoVM uses to manage data transfers between the host and the device. TornadoVM also provides a caching mechanism to avoid unnecessary data transfers. If the data on the device is up-to-date, TornadoVM can skip the data transfer and directly use the cached data on the device.

- **Execution engine (bytecode interpreter + OpenCL +PTX + SPIR-V/LevelZero drivers):** Instead of generating bytecode for execution on the accelerator, TornadoVM uses the bytecode to orchestrate the entire application from the host side. This approach, while an implementation detail, offers extensive possibilities for runtime optimizations, such as dynamic task migration and batch processing, while maintaining a clean design.[18]
- **JIT compiler + memory management:** TornadoVM extends the Graal JIT compiler to generate code optimized for GPUs and other accelerators. Recent changes in memory management mean each Java object now possesses its individual memory buffer on the target accelerator, mirroring Aparapi's approach. This shift enhances the system's flexibility, allowing multiple applications to share a single GPU, which is ideal for cloud setups.

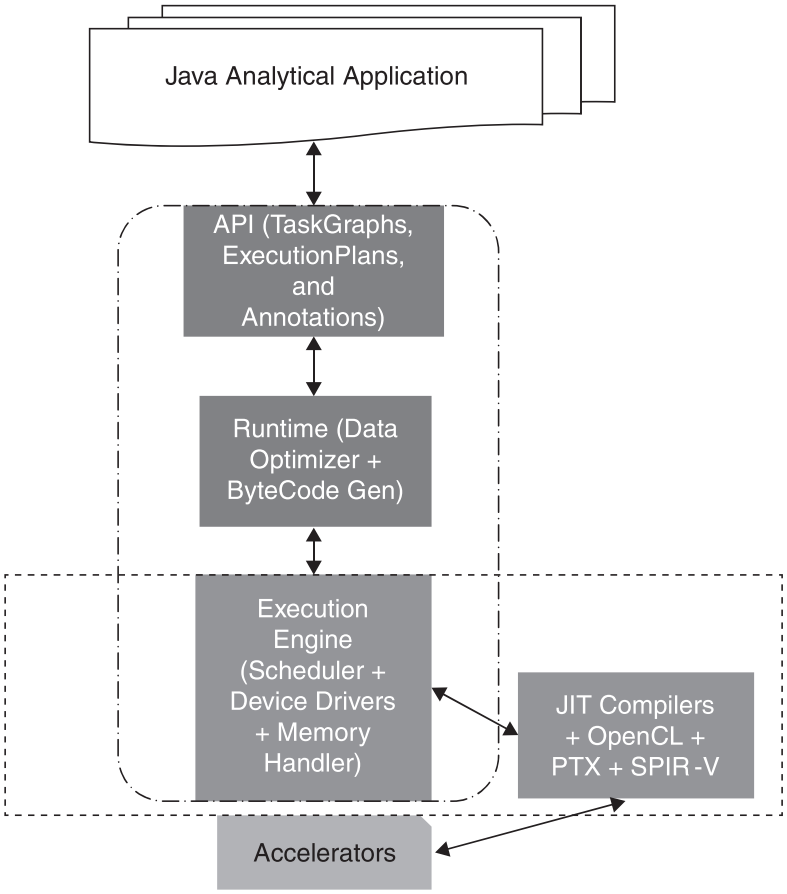

Figure 9.4 TornadoVM

[18] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. "Dynamic Application Reconfiguration on Heterogeneous Hardware." In *Proceedings of the 15th ACMSIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*, April 14, 2019. https://jjfumero.github.io/files/VEE2019_Fumero_Preprint.pdf.

In a traditional JVM, Java bytecode is executed on the CPU. However, with TornadoVM, Java methods that are annotated as tasks can be offloaded to the GPU or other accelerators. This allows TornadoVM to boost the performance of JVM-based applications.

Challenges and Future Directions

TornadoVM, like any pioneering technology, has faced its share of hurdles. One of the key challenges is the complexity of mapping Java's memory model and exception semantic to the GPU. Dr. Fumero emphasizes that TornadoVM shares certain challenges with Aparapi, such as managing nonblocking operations on GPUs due to the garbage collector. As with Aparapi, data movement between the CPU and accelerators can be challenging. TornadoVM addresses this challenge with a data optimizer that minimizes data transfers. This is possible though the Task-Graph API, which can accommodate multiple tasks, with each task pointing to an existing Java method.

TornadoVM continues to evolve and adapt, pushing the boundaries of what's possible with the JVM and exotic hardware. It serves as a testament to the potential benefits of offloading computation to the GPU and underscores the challenges that need to be surmounted to realize these benefits.

Project Panama: A New Horizon

Project Panama has made significant strides in the domain of Java performance optimization for specialized hardware. It is designed to enhance the JVM's interaction with foreign functions and data, particularly those associated with hardware accelerators. This initiative marks a notable shift from the conventional JVM execution model, historically centered on standard processors.

Figure 9.5 is a simplified block diagram showing the key components of Project Panama—the Vector API and the FFM API.

> **NOTE** For the purpose of this chapter, and this section in particular, I have used the latest build available as of writing this book: JDK 21 EA.

The Vector API

A cornerstone of Project Panama is the Vector API, which introduces a new way for developers to express computations that should be carried out over vectors (that is, sequences of values of the same type). Specifically, this interface is designed to perform vector computations that are compiled at runtime to optimal vector hardware instructions on compatible CPUs.

The Vector API facilitates the use of Single Instruction Multiple Data (SIMD) instructions, a type of parallel computing instruction set. SIMD instructions allow a single operation to be performed on multiple data points simultaneously. This is particularly useful in tasks such as graphics processing, where the same operation often needs to be performed on a large set of data.

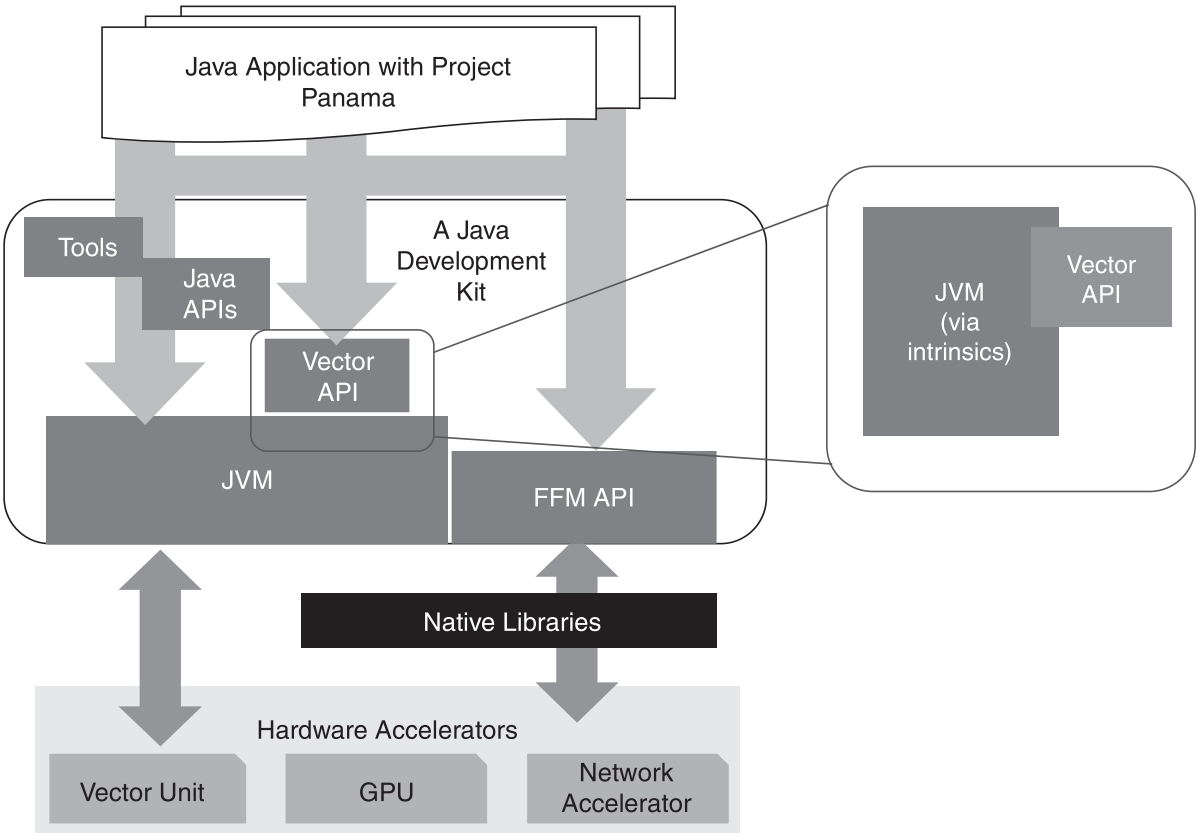

Figure 9.5 Project Panama

As of this book's writing, the Vector API resides in the `jdk.incubator.vector` module in the Project Panama early-access builds. It provides several benefits:

- **Vector computation expression:** The API offers a suite of vector operations that can be employed to articulate computations over vectors. Each operation is applied element-wise, ensuring uniformity in the processing of each element in the vector.
- **Optimal hardware instruction compilation → enhanced portability:** The API is ingeniously designed to compile these computations at runtime into the most efficient vector hardware instructions on applicable architectures. This unique feature allows the same Java code to leverage vector instructions across a diverse range of CPUs, without necessitating any modifications.
- **Leveraging SIMD instructions → performance boost:** The API harnesses the power of SIMD instructions, which can dramatically enhance the performance of computations over large data sets.
- **Correspondence of high-level operations to low-level instructions → increased developer productivity:** The API is architected in such a way that high-level vector operations correspond directly to low-level SIMD instructions via intrinsics. Thus, developers can write code at a high level of abstraction, while still reaping the performance benefits of low-level hardware instructions.

- **Scalability:** The Vector API offers a scalable solution for processing vast amounts of data in a concurrent manner. As data sets continue to expand, the ability to perform computations over vectors becomes increasingly crucial.

The Vector API proves particularly potent in the domain of graphics processing, where it can swiftly apply operations over large pixel data arrays. Such vector operations enable tasks like image filter applications to be executed in parallel, significantly accelerating processing times. This efficiency is crucial when uniform transformations, such as color adjustments or blur effects, are required across all pixels.

Before diving into the code, let's clarify the `factor` used in our example. In image processing, applying a filter typically involves modifying pixel values—`factor` represents this modification rate. For instance, a `factor` less than 1 dims the image for a darker effect, while a value greater than 1 brightens it, enhancing the visual drama in our *Shrek: The Musical* game.

With this in mind, the following example demonstrates the application of a filter using the Vector API in our game development:

```
import jdk.incubator.vector.*;

public class ImageFilter {
    public static void applyFilter(float[] shrekPixels, float factor) {
        // Get the preferred vector species for float
        var species = FloatVector.SPECIES_PREFERRED;
        // Process the pixel data in chunks that match the vector species length
        for (int i = 0; i < shrekPixels.length; i += species.length()) {
            // Load the pixel data into a vector
            var musicalVector = FloatVector.fromArray(species, shrekPixels, i);
            // Apply the filter by multiplying the pixel data with the factor
            var result = musicalVector.mul(factor);
            // Store the result back into the pixel data array
            result.intoArray(shrekPixels, i);
        }
    }
}
```

Through the code snippet shown here, `FloatVector.SPECIES_PREFERRED` allows our filter application to scale across different CPU architectures by leveraging the widest available vector registers, optimizing our SIMD execution. The `mul` operation then methodically applies our intended filter effect to each pixel, adjusting the image's overall brightness.

As the Vector API is evolving, it currently remains in the *incubator* stage—the `jdk.incubator.vector` package is part of the `jdk.incubator.vector` module in JDK 21. Incubator modules are modules that hold features that are not yet standardized but are made available for developers to try out and provide feedback. It is through this iterative process that robust features are refined and eventually standardized.

To get hands-on with the `ImageFilter` program and explore these capabilities, some changes to the build configuration are necessary. The Maven `pom.xml` file needs to be updated to ensure

that the `jdk.incubator.vector` module is included during compilation. The required configuration is as follows:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <source>21</source>
                <target>21</target>
                <compilerArgs>
                    <arg>--add-modules</arg>
                    <arg>jdk.incubator.vector</arg>
                </compilerArgs>
            </configuration>
        </plugin>
    </plugins>
</build>
```

The Foreign Function and Memory API

Another crucial aspect of Project Panama is the FFM API (also called the FF&M API), which allows a program written in one language to call routines or use services written in another program. Within the scope of Project Panama, the FFM API allows Java code to interoperate seamlessly with native libraries. This glue-less interface to foreign code also supports direct calls to foreign functions and is integrated with the JVM's method handling and linking mechanisms. Thus, it is designed to supersede the JNI by offering a more robust, Java-centric development model, and by facilitating seamless interactions between Java programs and code or data that resides outside the Java runtime.

The FFM API offers a comprehensive suite of classes and interfaces that empower developers to interact with foreign code and data. As of the writing of this book, it is part of the `java.lang.foreign` module in the JDK 21 early-access builds. This interface provides tools that enable client code in libraries and applications to perform the following functions:

- **Foreign memory allocation:** FFM allows Java applications to allocate memory space outside of the Java heap. This space can be used to store data that will be processed by foreign functions.
- **Structured foreign memory access:** FFM provides methods for reading from and writing to the allocated foreign memory. This includes support for structured memory access, so that Java applications can interact with complex data structures in foreign memory.

- **Life-cycle management of foreign resources:** FFM includes mechanisms for tracking and managing the life cycle of foreign resources. This ensures that memory is properly deallocated when it is no longer needed, preventing memory leaks.
- **Foreign function invocation:** FFM allows Java applications to directly call functions in foreign libraries. This provides a seamless interface between Java and other programming languages, enhancing interoperability.

The FFM API also brings about several nonfunctional benefits that can enhance the overall development experience:

- **Better performance:** By enabling direct invocation of foreign functions and access to foreign memory, FFM bypasses the overhead associated with JNI. This results in performance improvements, especially for applications that rely heavily on interactions with native libraries.
- **Security and safety measures:** FFM provides a safer avenue for interacting with foreign code and data. Unlike JNI, which can lead to unsafe operations if misused, FFM is designed to ensure safe access to foreign memory. This reduces the risk of memory-related errors and security vulnerabilities.
- **Better reliability:** FFM provides a more robust interface for interacting with foreign code, thereby increasing the reliability of Java applications. It mitigates the risk of application crashes and other runtime errors that can occur when using JNI.
- **Ease of development:** FFM simplifies the process of interacting with foreign code. It offers a pure-Java development model, which is more intuitive to use and understand than JNI. This facilitates developers in writing, debugging, and maintaining code that interacts with native libraries.

To illustrate the use of the FFM API, let's consider another example from the field of adaptive optics. Suppose we're working on a system for controlling the secondary mirror of a telescope, and we want to call a native function to adjust the mirror. Here's how we could do this using FFM:

```
import java.lang.foreign.*;
import java.util.logging.*;

public class MirrorController {
public static void adjustMirror(float[] secondaryMirrorAdjustments) {
        // Get a lookup object from the native linker
        var lookup = Linker.nativeLinker().defaultLookup();
        // Find the native function "adjustMirror"
        var adjustMirrorSymbol = lookup.find("adjustMirror").get();
        // Create a memory segment from the array of adjustments
        var adjustmentArray = MemorySegment.ofArray(secondaryMirrorAdjustments);
        // Define the function descriptor for the native function
        var function = FunctionDescriptor.ofVoid(ValueLayout.ADDRESS);
```

```
        // Get a handle to the native function
        var adjustMirror=Linker.nativeLinker().downcallHandle(adjustMirrorSymbol,function);
        try {
            // Invoke the native function with the address of the adjustment array
            adjustMirror.invokeExact(adjustmentArray.address());
        } catch (Throwable ex) {
            // Log any exceptions that occur
            Logger.getLogger(MirrorController.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

In this example, we're adjusting the secondary mirror using the `secondaryMirrorAdjustments` array. The code is written using the FFM API in JDK 21.[19] To use this feature, you need to enable preview features in your build system. For Maven, you can do this by adding the following code to your `pom.xml`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <compilerArgs>--enable-preview</compilerArgs>
                <release>21</release>
            </configuration>
        </plugin>
    </plugins>
</build>
```

The evolution of the FFM API from JDK 19 to JDK 21 demonstrates a clear trend toward simplification and enhanced usability. A notable change is the shift from using the `SegmentAllocator` class for memory allocation to the more streamlined `MemorySegment.ofArray()` method. This method directly converts a Java array into a memory segment, significantly reducing the complexity of the code, enhancing its readability, and making it easier to understand. As this API continues to evolve, further changes and improvements are likely in future JDK releases.

Challenges and Future Directions

Project Panama is still evolving, and several challenges and areas of future work remain to be addressed. One of the main challenges is the task of keeping pace with the ever-evolving hardware and software technologies. Staying abreast of the many changes in this dynamic field

[19] The FFM API is currently in its third preview mode; that means this feature is fully specified but not yet permanent, so it's subject to change in future JDK releases.

requires continual updates to the Vector API and FFM API to ensure they remain compatible with these devices.

For example, the Vector API needs to be updated as new vector instructions are added to CPUs, and as new types of vectorizable data are introduced. Similarly, the FFM API needs to be updated as new types of foreign functions and memory layouts are introduced, and as the ways in which these functions and layouts are used and evolved.

Another challenge is the memory model. The Java Memory Model is designed for on-heap memory, but Project Panama introduces the concept of off-heap memory. This raises questions about how to ensure memory safety and how to integrate off-heap memory with the garbage collector.

In terms of future work, one of the main areas of focus is improving the performance of the Vector and FFM APIs. This includes optimizing the JIT compiler to generate more efficient code for vector operations and improving the performance of foreign function calls and memory accesses.

Another area under investigation is improving the usability of the APIs. This includes providing better tools for working with vector data and foreign functions and improving the error messages and debugging support.

Finally, ongoing work seeks to integrate Project Panama with other parts of the Java ecosystem. This includes integrating it with the Java language and libraries, and with tools such as the Java debugger and profiler.

Envisioning the Future of JVM and Project Panama

As we stand on the precipice of technological advancement, the horizon promises that a transformative era for the JVM and Project Panama lies ahead. Drawing from my experience and understanding of the field, I'd like to share my vision for the future. Figure 9.6 illustrates one possible use case, in which a gaming application utilizes accelerators via the FFM and Vector APIs with the help of high-level JVM-language APIs.

High-Level JVM-Language APIs and Native Libraries

I anticipate the development of advanced JVM-language APIs designed to interface directly with native libraries. A prime example is the potential integration with NVIDIA's RAPIDS project, which is a suite of software libraries dedicated to data science and analytics pipelines on GPUs.[20] RAPIDS leverages NVIDIA's CUDA technology, a parallel computing platform and API model, to optimize low-level compute operations on CUDA-enabled GPUs. By developing JVM APIs that can interface with such native APIs, we could potentially simplify the development process, ensuring efficient hardware utilization. This would allow a broader range of developers—including those who may not have deep expertise in low-level hardware programming—to harness the power of hardware accelerators.

[20] "RAPIDS Suite of AI Libraries." *NVIDIA Developer*. https://developer.nvidia.com/rapids.

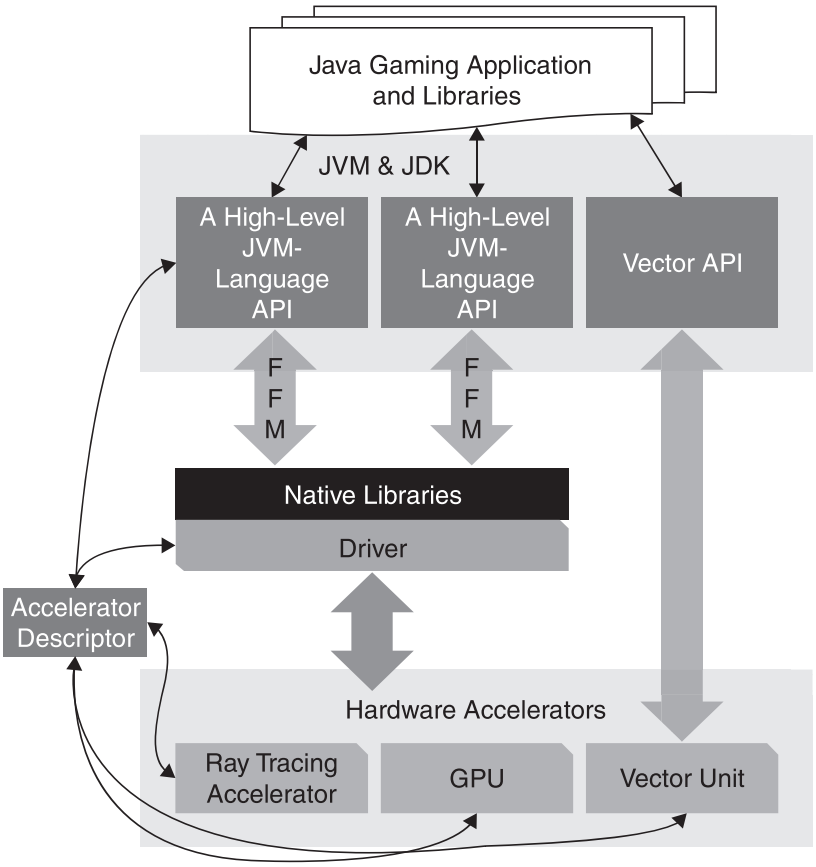

Figure 9.6 A Gaming Application Utilizing Accelerators via the FFM API and Vector API with the Help of High-Level JVM-Language APIs

Vector API and Vectorized Data Processing Systems

The Vector API, when synergized with vectorized data processing systems such as Apache Spark and vector databases, has the potential to revolutionize analytical processing. By performing operations on whole data vectors concurrently rather than on discrete data points, it can achieve significant speed improvements. This API's hardware-agnostic optimizations promise to further enhance such processing, allowing developers to craft efficient code suitable for a diverse array of hardware architectures:

- **Vector databases and Vector API:** Vector databases offer scalable search and analysis of high-dimensional data. The Vector API, with its platform-agnostic optimizations, could further scale these operations.
- **Analytical queries, Apache Spark, and Vector API:** Spark utilizes vectorized operations within its query optimizer for enhanced performance. Integrating the Vector API into this process could further accelerate the execution of analytical queries, capitalizing on the API's ability to optimize across different hardware architectures.

- **Parquet columnar storage file format:** The Parquet columnar storage file format for the Hadoop ecosystem could potentially benefit from the Vector API. The compressed format of Parquet files could be efficiently processed using vector operations, potentially improving the performance of data processing tasks across the Hadoop platform.

Accelerator Descriptors for Data Access, Caching, and Formatting

Within this envisioned future, accelerator descriptors stand out as a key innovation. These metadata frameworks aim to standardize data access, caching, and formatting specifications for hardware accelerator processing. Functioning as a blueprint, they would guide the JVM in fine-tuning data operations to the unique traits and strengths of each accelerator type. The creation of such a system is not just a technical challenge—it requires anticipatory thinking about the trajectory of data-centric computing. By abstracting these specifications, developers could more readily tailor their applications to exploit the full spectrum of hardware accelerators available, simplifying what is currently a complex optimization process.

To realize this vision, meticulous design and collaborative implementation are essential. The Java community must unite in both embracing and advancing these enhancements. In doing so, we can ensure that the JVM not only matches the pace of technological innovation but also redefines benchmarks for performance, usability, and flexibility.

This endeavor is about harmonizing the JVM's solid foundation with the capabilities of cutting-edge hardware accelerators, aiming for significant performance gains without complicating the platform's inherent versatility. Maintaining the JVM's general-purpose nature while optimizing for modern hardware is a delicate balance and a prime focus for Project Panama and the JVM community at large. With the appropriate developments, we are poised to enter a new epoch where the JVM excels in high-performance scenarios, fully leveraging the potential of modern hardware accelerators.

The Future Is Already Knocking at the Door!

As I ventured deeper into the intricate world of JVM and hardware acceleration, it was both enlightening and exhilarating to discover a strong synergy between my vision and the groundbreaking work of Gary Frost and Dr. Fumero. Despite our diverse research trajectories, the collective revelations at JVMLS 2023 showcased a unified vision of the evolution of this domain.

Gary's work with Hardware Accelerator Toolkit (HAT)[21] was a testament to this forward-thinking vision. Built on the foundation of the FFM API, HAT is more than just a toolkit—it's an adaptable solution across various environments. It includes the *ndrange* API,[22] FFM data-wrapping patterns, and an abstraction for vendor-specific runtimes to facilitate the use of hardware accelerators.

The *ndrange* API in HAT, drawing inspiration from TornadoVM, further enhances these capabilities. Complementing HAT's offerings, Project Panama stands out for its powerful functional API and efficient data management strategies. Considering that GPUs require specific data layouts, Project Panama excels in creating GPU-friendly data structures.

[21] www.youtube.com/watch?v=IbKBu3lTftc

[22] https://man.opencl.org/ndrange.html

Project Babylon[23] is an innovative venture emerging as a pivotal development for enhancing the integration of Java with diverse programming models, including GPUs and SQL. It employs code reflection to standardize and transform Java code, enabling effective execution on diverse hardware platforms. This initiative, which complements Project Panama's efforts in native code interoperation, signifies a transformative phase for Java in leveraging advanced hardware capabilities.

Another highlight of the discussion at JVMLS 2023 was Dr. Fumero's presentation on TornadoVM's innovative vision for a Hybrid API.[24] As he explained, this API seamlessly merges the strengths of native and JIT-compiled code, enabling developers to tap into speed-centric, vendor-optimized libraries. The seamless integration of Project Panama with this Hybrid API ensures uninterrupted data flow and harmonizes JIT-compiled tasks with library calls, paving the way for a more cohesive and efficient computational journey.

In conclusion, our individual journeys in exploring JVM, Project Panama, and hardware acceleration converge on a shared destination, emphasizing the immense potential of this frontier. We stand on the brink of a transformative era in computing, one in which the JVM will truly unlock the untapped potential of modern hardware accelerators.

Concluding Thoughts: The Future of JVM Performance Engineering

As we draw this comprehensive exploration of JVM performance engineering to a close, it's essential to reflect on the journey we've undertaken. From the historical evolution of Java and its virtual machine to the nuances of its type system, modularity, and memory management, we've traversed the vast landscape of JVM intricacies. Each chapter has been a deep dive into specific facets of the JVM, offering insights, techniques, and tools to harness its full potential.

The culmination of this journey is the vision of the future—a future where the JVM doesn't just adapt but thrives, leveraging the full might of modern hardware accelerators. Our discussions of Project Panama, the Vector API, and more have painted a vivid picture of what lies ahead. It's a testament to the power of collaboration and shared vision.

The tools, APIs, and frameworks we've discussed are the building blocks of tomorrow, and I encourage you to explore them while applying your own expertise, insights, and passion. The JVM community is vibrant, and ever evolving. Your contributions, experiments, and insights we collectively share will shape its future trajectory. Engage with this community. Dive deep into the new tools, push their boundaries, and share your findings. Every contribution, every line of code, every shared experience adds a brick to the edifice we're building.

Thank you for joining me on this enlightening journey.
Let's continue to explore, innovate, and shape the future of
JVM performance engineering together.

[23] https://openjdk.org/projects/babylon/

[24] Juan Fumero. *From CPU to GPU and FPGAs: Supercharging Java Applications with TornadoVM*. Presentation at JVM Language Summit 2023, August 7, 2023. https://github.com/jjfumero/jjfumero.github.io/blob/master/files/presentations/TornadoVM-JVMLS23v2-clean.pdf.