Main

Project Sumatra Wiki

This page, with its child pages, contains design notes for Project Sumatra

OpenJDK project page: http://openjdk.java.net/projects/sumatra
Repositories: http://hg.openjdk.java.net/sumatra/sumatra-dev/{scratch,hotspot,jdk,...} (repo info)
Developer list: http://mail.openjdk.java.net/mailman/listinfo/sumatra-dev

Goals

- Enable Java applications to take advantage of heterogeneous processing units (GPUs/APUs)
- Extend JVM JITs to generate code for heterogeneous processing hardware
- Integrate the JVM data model with data types efficiently processed by such hardware
- Allow the JVM to efficiently interoperate with high-performance libraries built for such hardware
- Extend the JVM managed runtime to track pointers and storage allocation throughout such a system

Challenges

Here are some of the specific technical challenges.

- mitigate the complexities of present-day GPU backends and layered standards
    - standards include: OpenCL, CUDA, Intel Phi, PTX, HSA HSA (forthcoming), ...
    - FIXME: choose 1-3 of the standards (e.g., PTX, HSAIL/HSA) for initial backend development
- build compromise data schemes for both the JVM and GPU hardware
    - define Java model for "value types" which can be pervasively unboxed (like tuples or structs)
    - need to support flatter data structures (Complex values, vector and RBGA values, 2D arrays) from Java
    - need to support mix of primitives and JVM-managed pointers
        - range of solutions: "don't"; like JNI array-critical; pinning read barrier; stack maps and safepoints in GPU
        - range of solutions: no pointers; pointers are opaque (e.g., indices into Java-side array); arena pointers; pinning read barrier.
    - need "foreign data interface" that is competent to interoperate (without copying) to standard sparse array packages
    - adapt (or extend if necessary) JNI as a foreign invocation interface that is competent to call purpose-built C code for complex GPU requests
- reduce data copying and inter-phase latency between ISAs and loop kernels
    - agreement of data structures will reduce copying
    - more flexible loop kernel container will allow loop kernel fusion
- cope with dynamically varying mixes of managed parallel and serial data and code
    - use JVM dynamic compilation techniques to build customized kernels and execution strategies
    - optimize computation requests relative to online data
- automatically (at each appropriate level of the system) sense load and distribute cleanly between CPU and GPUs
    - compile (online) JDK 8 parallel collection pipelines to data parallel compute requests
    - partition simple Java bytecode call graphs (after profile-directed inlining) into CPU and GPU
- learn to efficiently flatten nested or keyed parallel constructs
    - apply existing technology on nested data parallelism (to JVM execution of GPU code)
    - apply existing technology on MapReduce (to JVM execution of GPU code)
    - ensure that Java views of flattened and grouped parallel data sets are compatible with GPU capabilities
    - efficiently implement "nonlinear streams" in JDK 8 parallel collections
- create a practical and predictable story for loop vectorization, presumably user-assisted, and with useful failure modes
    - build a low-level library of vector intrinsics (e.g., AVX-style) that can be called (manually) from Java
    - apply existing technology for loop vectorization
    - build user-assisted loop vectorizers for Java, possibly based on type annotations (JSR 308)
- deal with exceptional conditions as they arise in loop kernels
    - allow GPU loop kernels to call back to CPU for infrequent edge cases (argument reduction, exceptions, allocation overflows, deoptimization of slow paths)
    - engineer a loop kernel container API which accounts for multiple CPU outcomes, and aggregates per kernel iteration (perhaps with continuation-passing style)
- define a robust and clear data-parallel execution model on top of the JVM bytecode, memory, and thread specifications
    - interpret (or adapt if necessary) the Java Memory Model (JSR 133) to the needs of data parallel programming
    - interpret (or adapt if necessary) the thread-based Java concurrency model (define GPU kernel effects in terms of bytecode execution by weakened quasi-threads)
- Investigate use of Java Language constructs and programming idioms that can be effectively compiled for a data-parallel execution engine (such as a GPU).
    - potential candidate - Lambda methods and expressions
    - other options?
- Investigate opportunities for GPU enabled 'intrinsics' versions of existing JDK APIs
    - candidates may be sort, (de)+compression, crc checking, search, convolutions etc.
- adopt and adapt insights from previous work on data-parallel Java projects
    - Fork/Join framework
    - Aparapi
    - Rootbeer
    - RIT Parallel Java
    - Terracotta

- jcuda - Java bindings for CUDA
- jocl - Java bindings for OpenCL
- jogamp-jocl - Jogamps' Java bindings for OpenCL
- FIXME: need a good list of references here

FIXME: Most of these items need their own wiki pages and/or email conversations

Roadmap

FIXME: In what order will we address these challenges?

Known investigations

FIXME: Add your work here!

See something wrong on this page? Fix it!

- Join sumatra-dev@openjdk.java.net
- Send an e-mail to sumatra-dev@openjdk.java.net:
  - Request editor rights to the "Project Sumatra" wiki
  - Include your Oracle SSO Username