Thread Swarms

Introduction: JVM vs. SIMD

The Java Virtual Machine represents parallel execution by threads. Each thread executes some bit of code on some bit of data. Both the code and the data are (in principle) independent from all other threads. At any moment, a thread performs a single work-unit consisting of an instruction, its inputs, and its outputs.

Parallel processing units execute similar work-units, with a difference: Several work-units may be executed simultaneously, sharing a single instruction and multiple parallel inputs and outputs.

A program that executes one ALU instruction across a parallel set of data inputs and outputs is called SIMD, single instruction multiple thread. If the data is structured as parallel sets of registers or stack temps, and the instructions are capable of performing control flow, then the model may be called SIMT, where the last word is "thread".

It is a useful exercise, and may be a useful programming model, to align these two models. Doing so will allow Java programs (perhaps with suitable compromises or restrictions) to render their execution as a series of SIMT operations on parallel processing units.

(Note: Highly regular computations, such as dense matrix arithmetic, will benefit independently from vectorization and tiled data layouts. SIMT modes of execution may or may not add value to these very regular problems.)

What's in a Java Thread?

At any given moment, a thread consists of:

- a bytecode T.bc being executed (part of a basic block within some "current method")
- local variables T.L[], expression stack T.S[], and monitors T.M[] (appropriate to the current method)
- a stack frame of all of the above T.F = < bc,L,S,M >
- a control stack of pending executions of either bytecode methods or native methods: T.C = {F(j)}
- thread-local values (accessed by java.lang.ThreadLocal), T.TL[]
- a permanently associated object of type java.lang.Thread, T.Thread

All of the above items may be referred to as the thread's *continuation*, T.cont. The idea behind that term is that those values, collectively, determine what the thread will do next if it is allowed to continue. It is useful to stop a thread an inspect its continuation. (That is what debuggers do.) It is even sometimes useful to force a thread to throw away its current continuation and execute a new one. (There is no way to express this in Java.)

It is also important to note what is *not* in a thread:

- data structures on the heap
- metadata describing types and methods
- bytecode and compiled code

A thread can be viewed as communicating as a peer with the heap, the metadata, and the code. For exmaple, a getfield instruction asks the heap to produce a value stored in a given object. The object is not inside the thread executing the instruction. The thread only holds a reference to the object. The instruction also consults the system metadata to verify that the desired field exists, and find out how to fetch it.

Java threads share memory structures but keep their instruction streams private. Parallel processing hardware such as GPUs can have a complementary design, with shared instruction streams but private memory. Localized, unshared memory simplifies communication, while several execution units may share a single instruction stream to reduce cross-lane interactions and/or amortize costs of instruction issuing.

Thread Swarms

Imagine a *thread swarm* G as a group of Java threads T(i), with full Java semantics, which executes all the threads with a high degree of alignment.

When a thread swarm starts, all of its threads are positioned at an instruction which is about to call the standard method Thread.run.

At any point, G has a pending continuation for each live thread T(i). In order to express the alignment between threads in the swarm, we express G's continuations as a multi-map rather than a set, with the keys of the map being current instructions:

G.conts = { bc -> { T(i).cont where T(i).bc == bc } }

Let's call each value set in the multi-map an aligned thread set.

A thread swarm makes progress by picking a key bc from G.conts.keySet and running that instruction in SIMD fashion across the relevant parts of each T(i) in parallel.

For example, if bc is a particular iadd, then all the T(i) waiting at that iadd add their top two stack elements and replace them with the sum. The threads then update their continuations to make ready to execute the next instruction after the iadd, and the G.conts map is updated to represent this.

Control Flow

When a thread swarm executes an instruction, there may be a variety of outcomes:

- the instruction completes normally and falls through to the next instruction
- the instruction completes normally by branching to another instruction

- the instruction performs an explicit throw
- the instruction returns from the current method
- the instruction performs a virtual call
- the instruction throw a JVM-mandated exception (NPE, CCE, OOB, etc.)
- the instruction blocks on some synchronization barrier
- the instruction blocks waiting for a query to a global resource (heap, metadata, etc.)

Notice that these outcomes are not mutually exclusive. When two threads are aligned, but their instruction outcomes are different, we say they *diverge*. They will end up in different entries in the G.conts multi-map.

This variability of control flow is a strongly entrenched feature of the JVM. In order to execute a thread swarm efficienty on GPU hardware, threads must be managed to as to converge as much as possible. For example, consider this classic diamond-shaped program:

p=a(); L:{ if (p) b(); else c(); } d();

If all threads start off at a() but they compute differing values of the boolean, some will execute b() and others c(). To begin to control divergence, no thread should execute d(), until all threads have left the if-then-else statement (the block labeled L).

(Note the consequence that throwing an exception or returning from b() or c() counts as exiting the block, even though the thread will not rejoin the others at d().)

Walking the Control Graph

We want to encourage convergence, to use parallel hardware more efficiently, and also to make less opportunity for race conditions.

The following rules can probably help:

- when an aligned set of threads is executed, no runnable threads are left behind
- empty aligned sets are not executed
- when there are threads runnable at more than one location, the earlier one is activated first
- expensive or complex "stragglers" may be referred back to the CPU for non-parallel execution
- backward branches (loops) may be special-cased, by blocking their threads until other threads have had a chance to branch backward also

A thread group may use more than one physical thread (or GPU array) to execute. For example, in an if-then-else statement, one processor might execute the "then" part which another deals with the the "else" part. Or, an implementation may choose to use a single physical processor, multiplexing the execution of the two parts.

Other Considerations

The SIMT model seems to allow some kinds of very serialized Java code to operate efficiently on GPUs.

The thread-swarm model is useless if the implementation actually requires that we create a large number of conventional threads. Perhaps a subclass of Thread would help express the necessary information to the system.

This model does not help much with the crucial task of inter-task communication. Normally, threads communicate with shared memory protected by synchronization. This will work in the SIMT model, but will be awkward overkill for simple operations like permutations or scans.

It should be possible to "roll up" or gather the final results (or provisional results) of a thread swarm computation. The rolled-up value must be structured as a map from processor to value, or a collection of values.

It should also be possible to "roll down" or scatter new values into the thread swarm computation from a map or collection.