Paul Holmes, pwh2125
GPU Computing, COMS W4995
Term Project Report

Jiminy – An Extension to the Geppetto Programming Language

I'm very interested in artificial intelligence, so any time I have a project to do for some class, I always try to do something in the AI domain. For this class, I chose to do an extension to an AI-oriented language I wrote for my Programming Languages and Translators class last year. The language is called Geppetto, named for the puppeteer from the movie *Pinocchio*. The purpose of Geppetto is to be "a language for the modeling of intelligent agents" – to bring a lifeless program to life, so to speak. At least, that's the theory. That claim turned out to be a *little* grandiose considering what the language ultimately ended up being able to do, but I like to think that the potential is still there.

In a very condensed nutshell, the premise of the language is basically to model a system, then "wind it up, let it go, and see what happens". More specifically, Geppetto allows the programmer to define objects called Entities and the Rules that operate on them. Each time unit (called a Cycle), the language evaluates all the Rules, potentially changing the states of the Entities, until some predefined end state is reached or a certain number of Cycles have been executed.

The problem – or one of them, anyway – was that the evaluation of the Rules each Cycle could become extremely time-consuming. While a given Rule *may* refer to just one specific Entity, that undercuts the purpose of the language. The real power of Geppetto is unleashed when Rules are generic and can thus refer to a whole class of Entities. So for example, in a system with 20 Entities, a Rule that refers to three Entities might have to be evaluated $20^3 = 8000$ times, once for each possible combination of Entities. That's exponential growth, which as any computer scientist knows can quickly become computationally infeasible. Ultimately I ended up putting restrictions on the language that made it possible to simulate only the most trivial of systems.

Solving the Problem

Jiminy (named for the helpful cricket in *Pinocchio*) is intended to help alleviate that problem. The idea behind Jiminy is that it can use the GPU to perform the Rule calculations in parallel, thus decreasing processing time.

Each Rule in Geppetto is a logical expression coupled with a statement to execute if the expression evaluates to true. As originally formulated, the logical expressions in a Rule are completely arbitrary – any statement that can be formed with the syntax of the language is legal provided it evaluates to a boolean. But to standardize things a little, and to make the problem more interesting, I decided to choose a particular data structure for the boolean condition in a Rule: the decision list.

For the uninitiated, a decision list is an ordered sequence of nodes, each of which has a condition which, if true, selects a value for the whole expression, and subsequent nodes are not evaluated. In other words, it's basically a sequence of if-then-else statements, with a final else as the default condition. It's a classic data type often used to demonstrate limits on computational complexity. Decision lists are very flexible and allow for much more complex expressions than a simple logical comparison.

Keeping it Simple

In theory, we could use any arbitrary logical expression in the condition of each node of the decision list, but to keep things simple I used randomly-generated "if variable == value" expressions, with the understanding that we could use any arbitrary logical expressions in their place if we so chose. More on this below.

Another simplification I imposed is that I didn't try to integrate Jiminy directly with the Geppetto language. That would have required a significant rewrite of Geppetto's Rule-processing engine, which I decided was well beyond the scope of this project.

Logical expressions in most programming languages (including Geppetto) may of course include variables. I had originally intended to send the symbol table to the GPU and do the evaluation of variables there, but due to time constraints I decided to do variable lookups on the host side instead. If more time were available, that's the first thing I would implement.

Keeping It Interesting

The problem I'm solving is simpler (and less glamorous) than many of the problems my classmates are solving in their projects, so to keep it interesting I added a couple of elaborations. First and foremost, I made the mechanism for sending logical expressions to the GPU as generic as possible. As noted above, the condition in each node of each decision list is currently limited to the form "if variable == constant". With that foreknowledge I *could* have encoded the expressions extremely compactly, especially if I made the assumption that the data types in question were all numeric. But that would have been an unrealistic assumption – in a real Geppetto program, there would be all kinds of expressions, including many that operate on strings. Therefore I wrote my encoding system such that it can be extended to encode *any* logical expression that can be expressed by the Geppetto language. In other words, by extending the framework I've put in place, one could use any arbitrary logical expression in the decision list nodes, instead of just the simple logical comparisons the code supports right now. As a result, the encoded expressions are more verbose than you might think is strictly necessary, and the code to interpret them is a **lot** more complex. But keep in mind that this is intentional.

As you can infer from the previous statement, Jiminy is implemented in Java. That was partially because I'm most comfortable coding in Java, but also because that's the language in which Geppetto is implemented. Since I'm using NVIDIS's CUDA API to interface with the GPU, it also means I had to use CUDA's Java bindings, which actually turned out to be not so bad compared to the other third-party libraries I've used in the past. It's clearly just a thin wrapper around the C language CUDA libraries (as evidenced by its rather awkward "Pointer" classes), but it is at least exceptionally well-documented and, within its limits, seems to be fairly robust and well-coded. Because I wrote my own kernel (as opposed to using NVIDIA's prepackaged kernels) I had to use the JCuda Driver API, which is one step closer to the hardware than the Runtime API that was discussed in class, but the differences are trivial. Nevertheless, JCuda was not without its challenges. In particular, it was quite tricky to figure out how to send arrays of strings to the device from Java – you end up sending a pointer to a pointer to a list of pointers!

The other main elaboration I added was making the kernel execution engine generic. Using inversion of control with an abstract base class, I tried to make it simpler to write an arbitrary kernel in Java: all the boilerplate code to initialize CUDA, get the available devices, etc., has all been hidden in the abstract base class, and all your kernel class has to deal with is the allocation and copying of memory back and forth. Of course, each function call you hide away in a base class like that requires an additional assumption to be made. For example, this code assumes you aren't using streams or shared memory, and that your grid is only 1-dimensional. So it has limited utility. But it does, at the least, isolate the "app-specific" parts of the code from the generic parts, which makes it a little easier to see what's going on.

Results

As I coded this project it became increasingly apparent that it was not going to be efficient to execute it on a GPU. For one thing, each expression has to be encoded as a string before transmission to the GPU. (There's no way to send it in "native" data types because the structure of each expression is not known ahead of time. So for instance, you can't sent an array of floats to the GPU when some the expressions contain integers, strings or booleans instead.) The time it takes to encode the expressions alone is probably comparable to the amount of time it would take to simply evaluate them on the host, so we're behind in the game before we even send the expressions to the GPU. Then the expressions have to be decoded on the GPU, evaluated, sent back, and the results interpreted. So long before I was finally able to execute an end-to-end test, I realized there was no way it was going to end up being more efficient to evaluate the expressions on the GPU.

Plus, there's the fact that each expression is different and possibly very complex, so their evaluation is going to be a totally divergent operation. That is, every thread on the GPU is going to be doing something different. As we know, that doesn't make for an efficient GPU program.

Concrete measurements bear out my misgivings. On my home PC, the evaluation of 1000 randomly-generated expressions takes about 600 on the GPU but only 3 or 4 ms on the CPU. (I was unable to evaluate more on the GPU because I ran into memory issues. So another item on my to-do list for this project given more time would be to implement streaming, or some other mechanism to alleviate the memory issues.)

In any case, evaluating logical expressions in parallel is clearly not a good candidate for implementation on a GPU, at least not as I have implemented it. Well, not every experiment is a success, right? I certainly learned a lot about the capabilities and limitations of GPU programming during the course of this project, so in that sense at least, one could still say it was a successful project.

Implementation & Execution Notes

The code was developed and tested on a 64-bit Windows system running Java 1.7.0.21. JCuda version 0.5.5 was used, and the code was compiled to CUDA architecture version 2.1 (for recursive function support). Eclipse Juno was the development IDE used.

JCuda requires host-specific files to execute at runtime. The necessary runtime files are included in the zip file's root directory. Although only one set of files is required for any given platform (in my case, the 64-bit Windows files), for convenience the JCuda runtimes for all available platforms are provided. Other supported platforms include 32-bit Windows, and 32- and 64- bit Linux. The JCuda website claims that the JCuda source code may be compiled on a wide variety of other platforms, though binaries for those platforms are not provided.

A JCuda jar file is also required. This file is present in the zip's file's root directory.

The latest version of the project's jar file is also included in the zip file's root directory, though of course a new version of the jar may be compiled from the provided source files at any time if desired. Since an IDE was used to build the project, there is no Ant or other build file provided, but there is nothing special about how the project is built – simply compile the Java files and zip them into a jar.

The host code, which is written in Java, ***is not*** in the root directory of the zip file submitted, because Java files are organized into packages that require their separation into different directories (for any well-designed project, anyway). The instructor's permission was obtained for this.

The kernel code, which is written in C, ***is*** present in the zip file's root directory. That code is automatically compiled at runtime for the current platform. The source file is called decisionListKernel.cu and the generated file is called decisionListKernel.ptx.

To execute the program, execute the following command line:

```
java –jar jiminy.jar
```

The results are not exactly spectacular: it will simply randomly generate a number of logical expressions, and then evaluate them on both the host and the GPU, giving timings for each. A number of trace statements print the application's progress to the console as it executes.