Rootbeer: Seamlessly using GPUs from Java

Philip C. Pratt-Szeliga
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY, USA
pcpratts@syr.edu

James W. Fawcett
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY, USA
jfawcett@twcny.rr.com

Roy D. Welch
Department of Biology
Syracuse University
Syracuse, NY, USA
rowelch@syr.edu

***Abstract* - When converting a serial program to a parallel program that can run on a Graphics Processing Unit (GPU) the developer must choose what functions will run on the GPU. For each function the developer chooses, he or she needs to manually write code to: 1) serialize state to GPU memory, 2) define the kernel code that the GPU will execute, 3) control the kernel launch and 4) deserialize state back to CPU memory.**

Rootbeer is a project that allows developers to simply write code in Java and the (de)serialization, kernel code generation and kernel launch is done automatically. This is in contrast to Java language bindings for CUDA or OpenCL where the developer still has to do these things manually.

Rootbeer supports all features of the Java Programming Language except dynamic method invocation, reflection and native methods. The features that are supported are: 1) single and multi-dimensional arrays of primitive and reference types, 2) composite objects, 3) instance and static fields, 4) dynamic memory allocation, 5) inner classes, 6) synchronized methods and monitors, 7) strings and 8) exceptions that are thrown or caught on the GPU. Rootbeer is the most full-featured tool to enable GPU computing from within Java to date.

Rootbeer is highly tested. We have 21k lines of product code and 6.5k lines of test cases that all pass on both Windows and Linux. We have created 3 performance example applications with results ranging from 3X slow-downs to 100X speed-ups.

Rootbeer is free and open-source software licensed under the GNU General Public License version 3 (GPLv3).

Keywords: GPGPU, Java, Compilers

I. INTRODUCTION

Graphics Processing Units (GPUs) are specialized processors separate from a CPU that offer hardware acceleration for specific types of computational problems. GPUs can accelerate a computation because there are many more ALUs acting in parallel in a GPU than a general purpose CPU. To use an Nvidia GPU (we focus on Nvidia brand GPUs because they offer recursive functions), a developer usually writes the application without parallelism in C or C++. Then, if the developer wishes to use a GPU and obtain a speedup, he or she must search for places in the code where there exists massive parallelism. Parallelism on the order of ten or one hundred threads is often not enough to accelerate and application using a GPU; there must be thousands of threads that will be run in parallel on the GPU in order to get a speedup. This is because each GPU thread is not as powerful as a standard CPU thread. GPU cores do not support complicated branch prediction or out-of-order execution and several execution units often share instruction fetch hardware. Since several execution units share instruction fetch hardware, on Nvidia devices, groups of 32 threads must all be executing the same parallel code, otherwise the threads will be executed serially. All of these things make a single GPU thread slower than a single CPU thread.

Converting a serial application to a parallel application using a GPU that gains a performance advantage is a complicated process. Most of the time, in general purpose code, finding the necessary parallelism required for GPU speedups is challenging. Often when this parallelism exists, it does not simply exist in a single loop. We have seen situations in scientific applications where there are many loops nested with intertwined function calls and serial code with an innermost loop. In the innermost loop there was enough parallelism needed to get a speedup from the GPU, but complicated code restructuring was needed to pull the innermost code out so that it could be executed on a GPU with one call to a function that launches the jobs.

Ideally when working in this situation a developer needs to be able to prototype different code restructurings to see which will get the best speedup. Using traditional tools, this process is quite error prone and time consuming because there are many manual steps.

Rootbeer aims to reduce the prototyping time needed to see performance improvements while using GPUs by eliminating many manual steps. Without Rootbeer, using Java language bindings, a developer must carefully convert complex graphs of Java objects into arrays of basic types. With Rootbeer this serialization is done automatically in a high performance way using all the available CPU threads on the processor. Without Rootbeer, a developer must write separate code in another language to specify what the GPU execution will do. With Rootbeer, the developer can simply write the GPU code in Java. Rootbeer does not use a modified Java syntax, so development can be done entirely in an existing IDE such as Netbeans or Eclipse. Rootbeer also has a native debugging mode where the GPU code is executed on a multi-core CPU inside a C++ debugger.

To demonstrate converting a C program to a GPU enabled CUDA [1] program we will use an application that sums the elements in a set of arrays. Then we will show the

same functionality, but with Java and Rootbeer. The original code in C is shown in Figure 1 below. It takes a 2-dimensional array of integer arrays at line 1 and returns a 1-dimensional array of summations of each integer array at line 9.

```
1  int * sumArrays(int ** arrays, int array_count, int size){
2    int * ret = (int *) calloc(sizeof(int)*array_count);
3    for(int i = 0; i < array_count; ++i){
4      int * curr_array = arrays[i];
5      for(int j = 0; j < size; ++j){
6        ret[i] += curr_array[j];
7      }
8    }
9    return ret;
10 }
```

Figure 1 – Original C code to sum arrays

To convert this code to CUDA using existing methods, we must implement a kernel function, allocate memory on the GPU, transfer data to the GPU memory, launch the GPU kernel function (with awareness of the size of the GPU and the size of the problem) and finally copy the data back from the GPU memory to the CPU memory. An example of this is shown in Figure 2 below. At line 5 we allocate GPU memory to hold the input set of arrays. Then in lines 7 through 12 we have to separately initialize and copy the state of each array in the input set. At line 13 we allocate space for the return array. At line 14 we call the gpuSum CUDA kernel that we have written manually (lines 25 to 34). At line 15 we copy the results back from the GPU memory to the CPU memory. Then in lines 17 through 21 we free all the memory we allocated on the GPU in this function. Finally at line 22 we return the resultant array containing the sums of the numbers in the arrays input variable. Clearly if we are doing this many different times and ways during our prototyping to get the best performance with even more complex inputs (such as object graphs) this can take a lot of work.

```
1  int * sumArrays(int ** arrays, int array_count, int size){
2    int * ret = (int *) calloc(sizeof(int)*array_count);
3    int ** d_arrays;
4    int * d_ret;
5    cudaMalloc((void **) &d_arrays,
6      sizeof(void *) * array_count);
7    for(int i = 0; i < array_count; ++i){
8      cudaMalloc((void **) &d_arrays[i],
9        sizeof(int) * size);
10     cudaMemcpy(d_arrays[i], arrays[i],
11       sizeof(int) * size, cudaMemcpyHostToDevice);
12   }
13   cudaMalloc((void **) &d_ret, sizeof(int) * size);
14   gpuSum<<<array_count, 1>>>(d_arrays, size, d_ret);
15   cudaMemcpy(ret, d_ret, sizeof(int) * size,
16     cudaMemcpyDeviceToHost);
17   for(int i = 0; i < array_count; ++i){
18     cudaFree(d_arrays[i]);
19   }
20   cudaFree(d_arrays);
21   cudaFree(d_ret);
22   return ret;
23 }
24
25 __global__ void gpuSum(int ** arrays, int size, int *
26 ret){
27   int tid = blockIdx.x;
28   int * curr_array = arrays[tid];
29   int sum = 0;
30   for(int j = 0; j < size; ++j){
31     sum += curr_array[j];
32   }
33   ret[tid] = sum;
34 }
```

Figure 2 – CUDA code to sum arrays in parallel

The next section gives an overview of how a developer would do the same thing with Rootbeer.

The rest of the paper is organized as follows: Section III discusses the high-level processing that Rootbeer does to an input java jar file. Section IV describes our high performance serializer that uses automatically generated Java Bytecode to get optimal performance. Section V describes our memory model where Java objects can be represented in a C style array of memory while still supporting all of the Java Programming Language features. Section VI provides details of how we cross-compile Java Bytecode into CUDA. Section VII explains how our tool dynamically and seamlessly fits a given GPU job into the available GPU memory at runtime. Section VIII provides a demonstration of the performance results of three test cases. Section IX compares Rootbeer to related work. Section X concludes our discussion of Rootbeer.

II. PROGRAMMING INTERFACE

To represent a kernel function (the code to be run on the GPU) in Java, the developer creates a class that implements the Kernel interface. This interface is shown in Figure 3 below. It has one method named "gpuMethod". The gpuMethod will be the main entry point of the developer's code on the GPU. The developer gets data onto the GPU by setting private fields in the class implementing the Kernel interface. Rootbeer automatically finds all reachable fields and objects from the gpuMethod and copies these to the GPU. Similarly, on the GPU results are saved to fields and Rootbeer will copy them back to the CPU.

```
1  public interface Kernel {
2    void gpuMethod();
3  }
```

Figure 3 – Kernel interface

Shown below in Figure 4, there is a class named ArraySum that implements the Kernel interface. It has three private fields that are used to transfer objects to/from the GPU. The first field is named source (line 2) and represents the array that this GPU thread will compute the sum of. When the developer creates the ArraySum object, a different source will be placed in each ArraySum to make each thread do different work. The second field is name ret (line 3) and it represents the array of return sum values. Each ArraySum object will share this

object. The final field named index (line 4) represents the index with which to put the array sum result into ret.

```
1  public class ArraySum implements Kernel {
2     private int[] source;
3     private int[] ret;
4     private int index;
5     public ArraySum (int[] src, int[] dst, int i){
6        source = src; ret = dst; index = i;
7     }
8     public void gpuMethod(){
9        int sum = 0;
10       for(int i = 0; i < array.length; ++i){
11          sum += array[i];
12       }
13       ret[index] = sum;
14    }
15 }
```

Figure 4 – Example of implementing Kernel interface

From lines 9 to 12 in Figure 4, the work of summing the array elements takes place. The bytecode for the gpuMethod method, as well as all methods accessible from it, is cross-compiled to CUDA so it can be executed on the GPU (section VI). Finally, the summation result is put in the ret array at a specified index (line 13). The ret array is shared among all GPU threads and a different index is passed into each ArraySum constructor (Figure 4, lines 5 through 7).

Once the Kernel class is created, a small amount of code is needed to setup the kernel objects and launch execution on the GPU. This is shown below in Figure 5.

In Figure 5 we have a sumArrays method that accepts a List of integer arrays that we are going to sum. First, we create a List<Kernel> to hold our kernels (line 3). Then we initialize the output array (line 4). From lines 5 to 7 we initialize each kernel object and keep track of it. Finally to launch the kernel on the GPU, lines 8 and 9 are used. We simply make an instance of Rootbeer and call runAll, giving the jobs as input.

```
1  public class ArraySumApp {
2     public int[] void sumArrays(List<int[]> arrays){
3        List<Kernel> jobs = new ArrayList<Kernel>();
4        int[] ret = new int[arrays.size()];
5        for(int i = 0; i < arrays.size(); ++i){
6           jobs.add(new ArraySum(arrays.get(i), ret, i));
7        }
8        Rootbeer rootbeer = new Rootbeer();
9        rootbeer.runAll(jobs);
10       return ret;
11    }
12 }
```

Figure 5 – Example of running GPU jobs

Once this code is saved and compiled into a resultant jar file, the developer runs the Rootbeer Static Transformer to produce the output jar. Then the developer uses the output jar just like he or she would use the input jar and it has GPU acceleration enabled. The command to run the Rootbeer Static Transformer is shown in the first paragraph of section III below. The rest of section III is dedicated to giving a high level overview of what processing the Rootbeer Static Transformer does.

III. HIGH LEVEL PROCESSING OVERVIEW

The Rootbeer Static Transformer is run with the command listed in Figure 6. InputJar.jar is a jar file to be processed and OutputJar.jar is the name of the resultant output jar.

```
1  $ java -jar Rootbeer.jar InputJar.jar OutputJar.jar
```

Figure 6 – Running the Rootbeer compiler

The Rootbeer Static Transformer starts by reading .class files from the input jar file and then uses the Soot Java Optimization Framework [13] to load the .class files into an intermediate representation in memory called Jimple [13]. Custom initialization code was made to initialize Soot so that memory consumption is minimized. The transformer finds each class in the input jar that implements the Kernel interface and treats the gpuMethod in each as an entry point. For each entry point, first it finds all of the types, methods and fields that will be used on the GPU. Then CUDA code is generated (see section VII for details) that can execute the computation on the GPU and compiled with nvcc into a cubin file. The cubin file is packed into the output jar so that is it accessible by the Rootbeer runtime during execution.

After CUDA code is generated, Java Bytecode is generated that can serialize all the types accessible from the current Kernel class. Section V discusses the associated performance with all the possible ways to serialize a Java Object and our chosen high performance design of the serializer.

After both CUDA and Java Bytecode generation is complete, the CompiledKernel interface is added to the class supporting the Kernel interface. This new interface allows the Rootbeer runtime to easily obtain the cubin code and a serializer object given an instance of the Kernel class.

Once these high level steps are complete, the Soot Java Optimization Framework converts the in memory class representations back to Java Bytecode and Rootbeer packs these .class files and the required Rootbeer runtime .class files into a resulting .jar file. The user of the compiled program then executes the resulting jar file as a normal Java application as shown in Figure 8.

```
1  $ java -jar OutputJar.jar <command_line_arguments>
```

Figure 8 – Executing a Java Application compiled with Rootbeer

The next four sections give more internal details of how Rootbeer operates. Performance and related work are then discussed in the final sections.

IV. HIGH PERFORMANCE (DE)SERIALIZATION IN JAVA

To create the highest performance (de)serialization to and from GPU memory, we studied the performance of every possible way of accessing fields in Java objects that will work with any Java virtual machine (we ignored adding direct memcpy calls into the Java Runtime because that is not portable). We read a byte from the same field 10,000,000 times using three approaches: 1) JNI, 2) Reflection and 3) Pure Java. The performance results are shown below in Figure 9. Everything that is serialized to the GPU is read from a field in some way. Since reading fields using pure java is the fastest method possible, the Rootbeer Static Transformer generates custom Java Bytecode that reads fields and places the results into a Java byte array. Then, once all the results are in the Java byte array, one JNI call is made to copy that byte array to GPU memory.

Method	Execution Time (ms)
JNI	247
Reflection	173
Pure Java	5

Figure 9 – Performance of Accessing a Java Field

V. REPRESENTING JAVA OBJECTS ON THE GPU

Java objects are represented with two segments in the GPU memory: static memory and instance memory. For each static field, in all types accessible from a root gpuMethod, a unique index into the static memory is calculated to represent where the value is stored. All values are aligned relative to the size of the primitive data type used to store the value (required on the GPU). Then when CUDA code is generated, those same offsets are used to read and write static fields.

Instance memory is represented as a set of objects, each with a 16 byte header. The header contents for objects and arrays are shown in Figure 10 and 11 below.

Description	Size
Reserved for Garbage Collector	10 bytes
Derived Type	1 byte
Created on GPU flag	1 byte
Object Monitor State	4 bytes

Figure 10 – Header Contents for Objects

The object header currently contains 10 bytes that are reserved for the garbage collector. A GPU garbage collector is an item in our future work. The derived type is used instead of a function pointer in a virtual function pointer table. Instead of this CUDA code with a switch statement is generated. The "Created on GPU" flag is used when an object is created with new on the GPU. The deserializer detects this and needs to execute a corresponding new in Java when transferring control back to the CPU.

The array header is the same as the object header, except it also contains the length of the array. Since any object is at least 16 bytes, this fact is used when representing references. References to objects are stored as 4 byte integers that are shifted right 4 bits. This allows us to address 68 gigabytes of memory using only an integer.

Description	Size
Reserved for Garbage Collector	6 bytes
Derived Type	1 byte
Created on GPU flag	1 byte
Object Monitor State	4 bytes
Length of Array	4 bytes

Figure 11 – Header Contents for Arrays

After the header, in objects, all of the fields that point to other objects are stored in a block and grouped together as 32-bit integers (this is to support the future garbage collector). After the pointers, the value types are stored starting from the largest type to the smallest to efficiently pack value types while maintaining alignment. Data for primitive, single dimensional array types are stored directly after the header and can be indexed easily because the size of each element is known. The data stored in multidimensional arrays for each array element is a reference to another array with one less dimension (but an array of one dimension is stored directly). Similarly, arrays of object types store references to the actual objects in each array element.

VI. CUDA CODE GENERATION

Java Bytecode is converted to the CUDA programming language by directly analyzing Jimple statements from Soot. For each field that is accessed on the GPU a getter and setter function is generated that contains hard-coded constants to get to the right offset in memory from a base pointer. Polymorphic methods are implemented by generating an additional invoke function that switches to the correct concrete function based on the "Derived Type" stored in the GPU memory (see Figures 10 and 11). Java methods are represented on the GPU by encoding the class and method name into a function name. The GPU memory pointer is passed in as the first argument to every function and a "this pointer" is passed as the second argument. The original function arguments are passed in next, and a pointer to a long that represents exceptions are passed in last. Exceptions are supported by checking to see if the exception pointer is non-zero after every action that could cause a change in state. If the exception pointer is non-zero, generated CUDA code for the catch block is executed, or if there is none, execution bubbles up to the calling function.

Creating new objects is possible on the GPU in Rootbeer. To support this, a globally accessible pointer to the end of the allocated memory is maintained. This pointer is atomically incremented with the size of allocation and the old value is used as the returned reference from the GPU malloc. If the GPU runs out of memory, an OutOfMemoryException is bubbled back up the call stack, possibly all the way to the CPU.

Rootbeer supports re-entrant synchronized methods and locks on objects. A 32-bit integer in the object header represents the lock that is initialized to -1. A thread can enter the lock if it is -1 or already equal to the current thread id. To enter the lock, the current thread id is exchanged with the previous lock value. An example of a generated synchronized

method is shown in Figure 12. There is a while loop at line 6 that fools the CUDA static analyzer. Using just a regular while loop will cause deadlocks. Then code is inserted for the method during CUDA code generation at line 15. The generated code must be in the same function that acquired the lock or else a deadlock will occur.

```
1  void example_synch_method(){
2    char * mem = getReferenceToMonitorState()
3    int id = getThreadId();
4    int count = 0;
5    int old;
6    while(count < 100){
7      old = atomicCAS((int *) mem, -1 , id);
8      if(old != -1 && old != id){
9        count++;
10       if(count > 99){
11         count = 0;
12       }
13       continue;
14     } else {
15       //insert code for method here
16       if(old == -1){
17         atomicExch((int *) mem, -1);
18       }
19     }
20   }
21 }
```

Figure 12 – Example Synchronized Method

Java strings are supported on the GPU by converting string constants from Java Bytecode into actual java.lang.String objects allocated with new on the GPU.

The CUDA kernel function is named the same and contains the same parameter list for every root gpuMethod. This allows us to hardcode the encoded function name obtained from a ptx file into our JNI runtime that loads that function from a cubin file.

VII. EXECUTING JOB SETS THAT ARE LARGER THAN THE AVAILABLE GPU SIZE

To execute job sets that have more memory or thread requirements than what the current GPU has, the Rootbeer runtime will repeatedly serialize, execute and deserialize with a smaller subset until all threads are processed.

VIII. PERFORMANCE RESULTS

In addition to our correctness test cases, we have also made three test cases to demonstrate that Rootbeer can achieve a performance improvement over regular Java code. The first test case is a dense matrix multiplication, the second is an FFT that is done brute force and the third is a sobel filter.

Our tests were run under Ubuntu 10.04.3 LTS on a Dell Precision T7400 with one 4-core Xeon CPU running at 2GHZ and 25GB of ram. The GPU is an Nvidia Tesla C2050 with 3GB of ram.

1. Dense Matrix Multiplication

For the dense matrix multiplication test case we multiplied two matrices of size 4096x4096.

System	Time (ms)
Java Only	3,531,551
Java with Rootbeer	52,662

In this case Rootbeer is 67X faster. The table below lists the serialization time that Rootbeer requires.

Event	Time (ms)
Rootbeer Serialization	557
GPU Execution	51,687
Rootbeer Deserialization	418

2. BruteForce FFT

For the BruteForce FFT test case we calculated the real part of an FFT in an $O(n^2)$ manner rather than an O(nlgn). The size n was 114688.

System	Time (ms)
Java Only	4,714,566
Java with Rootbeer	87,255

In this case Rootbeer is 54X faster. The table below lists the serialization time that Rootbeer requires.

Event	Time (ms)
Rootbeer Serialization	15
GPU Execution	87,220
Rootbeer Deserialization	20

3. Sobel Filter

For the Sobel Filter test case we calculate the result of an edge detection kernel run over a 1600x1200 image. We use 1024 GPU threads for this computation.

System	Time (ms)
Java Only	129
Java with Rootbeer	502

In this case Rootbeer is 3.8X slower. As can be seen in the figure below, the serialization takes up much of the time.

Event	Time (ms)
Rootbeer Serialization	167
GPU Execution	125
Rootbeer Deserialization	210

This test case is also not as suited for the GPU because it has O(n) computations for a data size of O(n). The matrix multiplication has $O(n^3)$ and the FFT has $O(n^2)$. Our future work will try to address speeding up computations similar to the Sobel Filter by further optimizing serialization when there

are a large number of kernels and supporting shared memory usage on the GPU.

IX. RELATED WORK

There are several Java Language bindings (jCUDA [2], jocl [3] and JavaCL [4]). These projects still require a developer in Java to serialize complex object graphs to primitive arrays manually and write the kernel code in another language.

PyCUDA [5] helps a developer use CUDA from Python. It requires the developer to write the kernel code in CUDA and manually serialize from complex graphs of object in Python to arrays in CUDA. Again, Rootbeer does this serialization automatically and with high performance.

The PGI Accelerator Compiler [7] allows a developer to program in C or Fortran and use the GPU. The developer uses pragmas to specify what regions of code are to be used on the GPU. This compiler has a nice implementation but our goal was to support Java.

In Alan Chun-Wai Leung's thesis [8] he created an extension to the Jikes Research Virtual Machine (JikesRVM) that detects parallelism at runtime and automatically executes parallelism on the GPU if it is cost effective. Leung's work is a prototype that focuses on automatic parallelization while our work is production quality and focuses on high performance and reliable execution while ignoring automatic parallelization. Our software is also freely available and open source.

In Peter Calvert's dissertation [11] he created a compiler where developers can "indicate loops to be parallelized by using Java annotations". His implementation has known places where the compiler fails that Rootbeer does support and does not support all of the Java Programming Language features. Our only known places where the compiler fails are when using dynamic methods, reflection and native methods due to our usage of static type analysis and our lack of facility for reading the bodies of native methods.

Aparapi [12] has recently been released. This open source project is very similar to our work, but it is less full-featured. Aparapi only supports single dimensional arrays of primitive types while Rootbeer supports most of the Java Programming Language.

X. CONCLUSIONS

In this paper we have introduced Rootbeer, a system to seamlessly use the GPU from Java. Rootbeer is the most complete implementation of a system giving the ability to use the GPU from Java to date. It is production quality software that can achieve speedups and is ready to be used by researchers needing to accelerate Java code using GPUs.

Rootbeer can be possibly used for any application that is computational bound including simulations for physics, chemistry and biology. Rootbeer lowers the learning curve required to write GPU programs so scientists who are not also computer scientists may have an easier time using it over traditional methods to program GPUs such as CUDA or OpenCL.

XI. FUNDING

The equipment for our tests and development was partly purchased from National Science Foundation grant number MCB-0746066 to R.D.W.

XII. REFERENCES

[1] Nvidia CUDA: http://developer.nvidia.com/category/zone/cuda-zone

[2] jCUDA: http://www.jcuda.de/

[3] jocl: http://www.jocl.org/

[4] Ché zOlive. JavaCL: http://code.google.com/p/javacl/

[5] Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih. PyCUDA: GPU Run-Time Code Generation for High-Performance Computing

[7] Implementing the PGI Accelerator model. Michael Wolfe, ACM International Conference Proceeding Series; Vol. 425 archive. Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics. 2010

[8] Leung, A.C.-W.: Thesis: Automatic Parallelization for Graphics Processing Units in JikesRVM. University of Waterloo, Tech. Rep. (2008)

[9] Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 101–110. ACM, New York (2009)

[10] Nickolls, J., Buck, I., Garland, M., Nvidia, Skadron, K.: Scalable Parallel Programming with CUDA. ACM Queue 6(2), 40–53 (2008)

[11] Peter Calvert, Parallelisation of Java for Graphics Processors. http://www.cl.cam.ac.uk/~prc33/publications/2010-javagpu-diss.pdf

[12] Aparapi: http://code.google.com/p/aparapi/

[13] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon and Phong Co. Soot - a Java Optimization Framework. Proceedings of CASCON 1999. pages 125—135. www.sable.mcgill.ca/publications