Offloading Java to Graphics Processors

Peter Calvert (prc33@cam.ac.uk)

University of Cambridge, Computer Laboratory

Abstract

Massively-parallel graphics processors have the potential to offer high performance at low cost. However, at present such devices are largely inaccessible from higher-level languages such as Java. This work allows compilation from Java bytecode by making use of annotations to specify loops for parallel execution. Data copying to and from the GPU is handled automatically. Evaluation showed that significant speedups could be achieved (up to $180\times$), that were similar to those claimed by other less-automated work.

1 Problem and Motivation

It has been widely recognised that future improvements in processor performance are likely to come from parallelism rather than increased clock speeds [1]. As well as multi-core CPUs, massively-parallel graphics processors (GPUs) are now widely available that can be used for single instruction multiple data (SIMD) style general processing. However, existing frameworks for making use of this hardware require the developer to have a detailed understanding of the underlying architecture, and to make explicit data movements.

Therefore, while the scientific community are making extensive use of such hardware, it has been difficult to do so from standard desktop applications. These tend to be written in higher-level languages such as Java, and it is from these languages that the hardware has been largely inaccessible. However, while providing bindings [2] for the standard APIs (such as CUDA) makes the hardware available, it goes against many of the reasons for using a high-level language:

- High-level languages are typically used to make programs portable, so making use of hardware specific libraries loses this benefit.
- Computation kernels will still have to be written in the language supported by the framework (typically a C variant), leading to a multi-language solution.

If we could provide a solution that allowed high-level code to be executed on the GPU without affecting protability, then GPUs might be used in more 'everyday' applications. It is also important to minimise the burden on the developer. In this work, I concentrated on Java and NVIDIA's CUDA framework.

2 Background and Related Work

General purpose GPUs support execution of a single computation kernel across a grid—i.e. data parallelism. However, in order to make use of this, we must find such parallelism in our program. Normally, this will be in the form of (nested) `for` loops where each iteration is independent from the others. Unfortunately, standard Java code does not provide dependency information. Other work [3] has tried to reconstruct this data by analysis. However, automatic parallelisation is well known to be a hard problem due to the difficulty of alias analysis—i.e. determining whether two references might point to the same memory location.

(a) Single Entry/Single Exit   (b) Multiple Entries   (c) Multiple Exits
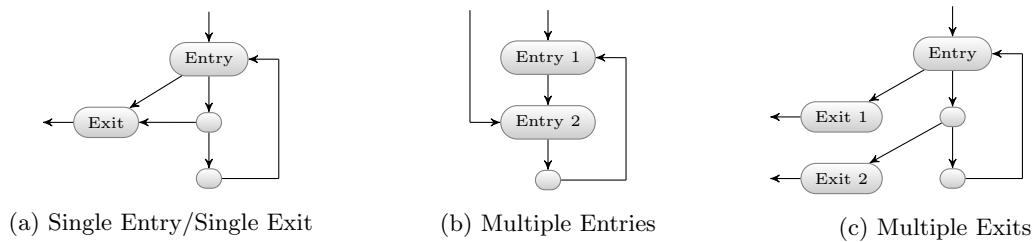
Figure 1: Various examples of loops.

A further complication in using graphics processors is that they operate on separate memory. Data must therefore be transferred to and from the device as appropriate. However, to avoid excess time overheads, we want to restrict these transfers to the minimum possible.

A common alternative has been to simply wrap the GPU API, either as a library [2] or by extra language syntax [4]. Even when the solution allows the kernel's source to be included as a string in the high-level source code, this approach suffers from the problems already described.

3   Approach and Uniqueness

Unlike other approaches, the compiler produced by this work neither relies on automatic analysis nor acts as a simple wrapper for a low-level API. Instead, I introduce program annotations that explicitly indicate parallel `for` loops. Based on this meta-data, the compiler determines the required data transfers by analysis. It also produces GPU code from the Java bytecode, rather than requiring kernels to be written in the CUDA language. This is all done as an extra compilation step, taking compiled class files and producing modified classes that make the required to calls to a native library (also produced by the compiler) that uses the GPU. Since this step acts on bytecode, it could be performed on an end-user machine without access to source code. This offers some portability, since the transformation can be made just when a suitable GPU is available (provided the required annotations are present).

Whilst [3] also acts on bytecode, this is done within the virtual machine. There are arguments for both approaches. Incorporating the extra steps into the virtual machine requires less end-user intervention, but means that the compilation must be done on every execution, and also that users must replace their existing virtual machine.

3.1   Loop Detection

Java bytecode is made up of *unstructured* control flow rather than constructs such as conditionals and loops. For the compiler, it is important to reconstruct at least the applicable loops to be able to convert sequential `for` loops into parallel ones. In general, loops can have a wide variety of forms as shown in Figure 1. A standard restriction is to those with only a single entry point—*natural loops*. These can be detected using a simple dataflow analysis that considers *dominators* [5, p655].

The class of loops that can be executed on the GPU must also be suitable for parallel execution. In this work, the loops addressed are limited to *trivial loops*, defined as natural loops for which there is only one exit, which compares the loop's index variable with an expression. All writes within the loop to the index variable must be either increments or decrements—i.e. it must be an *increment variable*. This is a slightly more general form of the loops considered in [6].

3.1.1   Increment Variables

Increment variables can be detected using a simple analysis. We maintain a value $I_v(x)$ for each variable v at each point x in the code under consideration (normally a loop body). This value is taken from the set of values $\mathbb{Z} \cup \top$. An integer $n \in \mathbb{Z}$ indicates that the net effect on the variable from entry until the given point is an increment of n. $\top$ indicates that the variable is written to in a more 'complex' fashion. We can calculate these values in an iterative manner using the following equation:

$$I_v(x) = \begin{cases} m + n & \text{if instruction } x \text{ increments } v \text{ by } n, \text{ and } \forall p \in \text{predecessors}(x).I_v(p) = m \\ \top & \text{otherwise} \end{cases}$$

A proof that this analysis converges is given in the dissertation associated with this work [7, Section 3.3.2].

3.1.2   `@Parallel` Annotation

When a developer indicates that a loop should be run in parallel using a `@Parallel` annotation, this means that (apart from increment variables) there are no *loop-carried dependencies*, and that the expression with which the loop index variable is compared is unaffected by the loop body. In this case, it is therefore possible to know before executing the loop, how many iterations will occur, and the values of the increment variables on each one.

3.2   Code Generation

Once it is established that a loop is suitable for GPU execution, it is necessary to compile the loop body for the graphics processor. This must be done before any alterations are made to the bytecode itself, since the body may make use of features not supported by the GPU (such as recursion or exceptions).

The compiler generates CUDA C, which is in turn passed to NVIDIA's compiler. This avoided the unnecessary effort of writing an optimiser. Java objects are mapped onto C `struct`s that are supported by CUDA directly. Methods within the objects are converted into simple functions by a name mangling process similar to that used by JNI [8, Table 2-1]. Unfortunately, it is not possible to support inheritance, since function calls are determined statically. In the cases of both arrays and objects, extra checks are needed to ensure that multiple copies of the same object are not imported (due to aliasing).

3.3   Determining Data Transfers

Finally, the compiler must replace the loop with a call to native code that first copies relevant data to the graphics card, and then invokes the generated kernel. The interesting aspect of this is determining which variables need to be *copied in* to the kernel, and which need to be *copied out*.

3.4   Copy-In

The variables used by the kernel can be determined by using *live variable analysis* [5, p608] on the kernel. Arrays can be handled by marking the whole array live whenever any element of it is either read or written to. This treatment is similar to the behaviour of a 'write allocate' memory cache (where a cache line must be loaded even for a memory write).

Numeric	Time (ms)			Speedup Factor	
$N = 1.5 \times 10^6$	*Copy-In*	*Execute*	*Copy-Off*	*Kernel Only*	*Overall*
CPU Execution	–	1180	–	–	–
Local Variable (1D)	9.3	6.2	15.2	190	38
Static Field (1D)	9.3	5.0	15.2	236	39
Object Array	11112	5.4	59.6	219	–
2D Array	25.5	5.2	34.0	227	18

Table 1: Speedup factors for simple numeric benchmarks.

Benchmark	**Series** (Floating Point)			**Crypt** (Integer)		
Data Size	10^4	10^5	10^6	$3 \cdot 10^6$	$2 \cdot 10^7$	$5 \cdot 10^7$
CPU (ms)	17971	182894	2878469	414	2190	5344
This Project on *Geforce GTX 260* (ms)	99	968	9358	41	245	545
JCUDA on *Tesla C1060* (ms)	110	1040	10140	20	160	450

Table 2: Comparison of Java Grande benchmark timings with JCUDA.

3.5   Copy-Out

Any writes made by the kernel must be to an array, since otherwise an output dependency would exist between different iterations of the loop. The above treatment of arrays means that the *copy-out* set is always a subset of the copy-in set. However, by using alias analysis it is possible to refine this result. The simple alias analysis used in the compiler does not always converge to a result. It is therefore occasionally necessary to revert to the complete copy-in set (for further details see [7, Section 3.3.3]).

4   Results and Contributions

4.1   Results

The compiler was applied to a range of benchmarks, consisting of both examples written to test specific aspects and also existing publically-available code.

The performance results were encouraging and suggested that the usual speedups associated with CUDA could still be achieved within Java. On the hardware used ($2\times$ 3.2GHz Pentium 4, with Geforce GTX 260), speedup factors of about $30\times$ were typical for the simple numeric benchmarks (calculating $\sin^2 x + \cos^2 x$ for each number in an array) as shown in Table 1. However, cases requiring many small, and potentially-aliased, objects to be transferred expose the communication and JNI overheads, exhibiting slow downs when little work is done per-object on the GPU. It is clear from the results that reducing data transfer is key to gaining further improvements.

Measurements were also made for two benchmarks from the Java Grande Benchmark Suite [9]. These were broadly similar to the results published for JCUDA [4] on similar hardware (see Table 2). The crypt benchmark is interesting since it shows that the benefits of GPU execution are not restricted to floating point computations.

A final benchmark used was that of Mandelbrot set computation. In this case, the number of iterations used in the computation allows the amount of time spent on the GPU to be varied without affecting the data transfer overheads. This confirmed the expected result that overall speedups were improved when the amount of GPU computation was large compared with the data movement required (Figure 2).
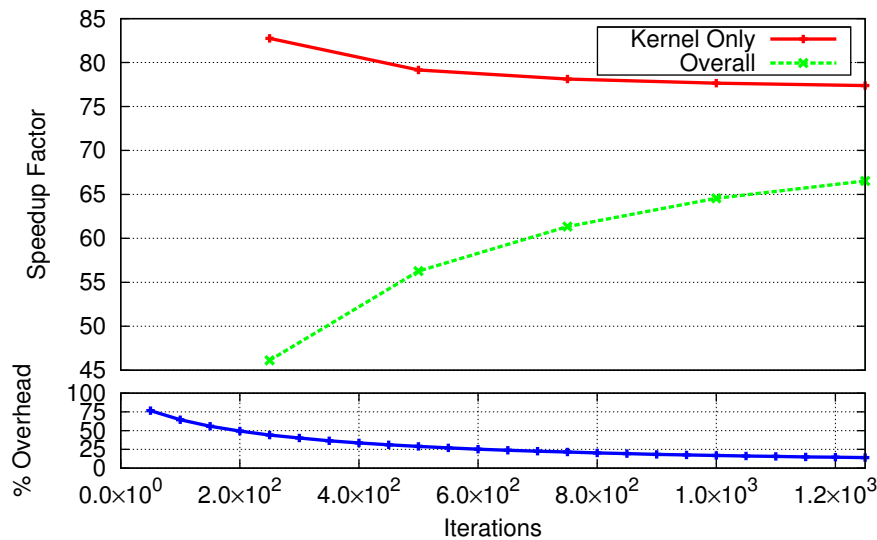
April 4, 2011

Figure 2: Mandelbrot set computation speedup factors, with varying number of iterations (grid fixed at 8000×8000).
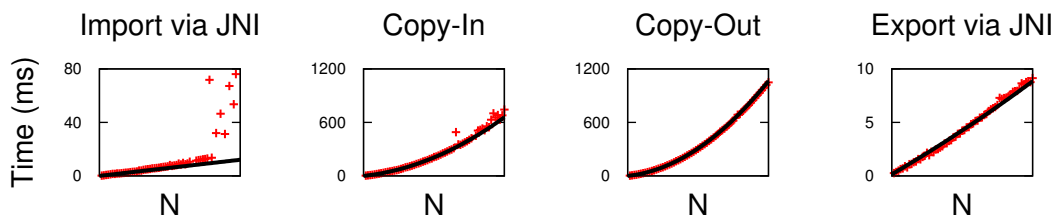

Figure 3: Fit of overheads model to Mandelbrot timings.

In addition, a model was developed that predicts the overheads for a variable given its type and length. Excluding cases with large numbers of objects, this gave accurate results and might allow future work to give feedback on the suitability of GPU execution, or even choose between CPU and GPU execution at runtime. Figure 3 shows the predictions made for the Mandelbrot set benchmark.

4.2   Conclusions

This work has developed a new compiler which enables Java code to be executed on massively-parallel graphics processors without any need for explicit data transfers. The results achieved demonstrate that it is possible to simplify the development process for making use of GPUs without sacrificing performance compared to other recent, but less automated, approaches.

It is also possible to predict the overheads in certain scenarios, and this might be important in future work that is either more automated or which gives more developer guidance.

4.3   Acknowledgements

This work was completed as a final-year undergraduate project at the University of Cambridge Computer Laboratory between October 2009 and May 2010, under the supervision of *Dr Andrew Rice* and *Dominic Orchard.*

April 4, 2011

References

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr Dobb's Journal*, March 2005.

[2] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA: GPU Run-Time Code Generation for High-Performance Computing," *Arxiv preprint arXiv:0911.3456*, 2009.

[3] A. Leung, O. Lhoták, and G. Lashari, "Automatic parallelization for graphics processing units," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, 2009, pp. 91–100.

[4] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, 2009.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers : Principles, Techniques, & Tools, Second Edition*, 2nd ed.   Addison-Wesley, 2007.

[6] A. Bik and D. Gannon, "`javab` - A prototype bytecode parallelization tool," in *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[7] Peter Calvert, "Parallelisation of Java for Graphics Processors," Final-year dissertation at University of Cambridge Computer Laboratory. Available from http://www.cl.cam.ac.uk/~prc33/, 2010.

[8] S. Liang, *Java Native Interface 6.0 Specification*.   Sun, 1999.

[9] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A methodology for benchmarking Java Grande applications," in *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*.   New York, NY, USA: ACM, 1999, pp. 81–88.