UNIVERSITY OF CAMBRIDGE

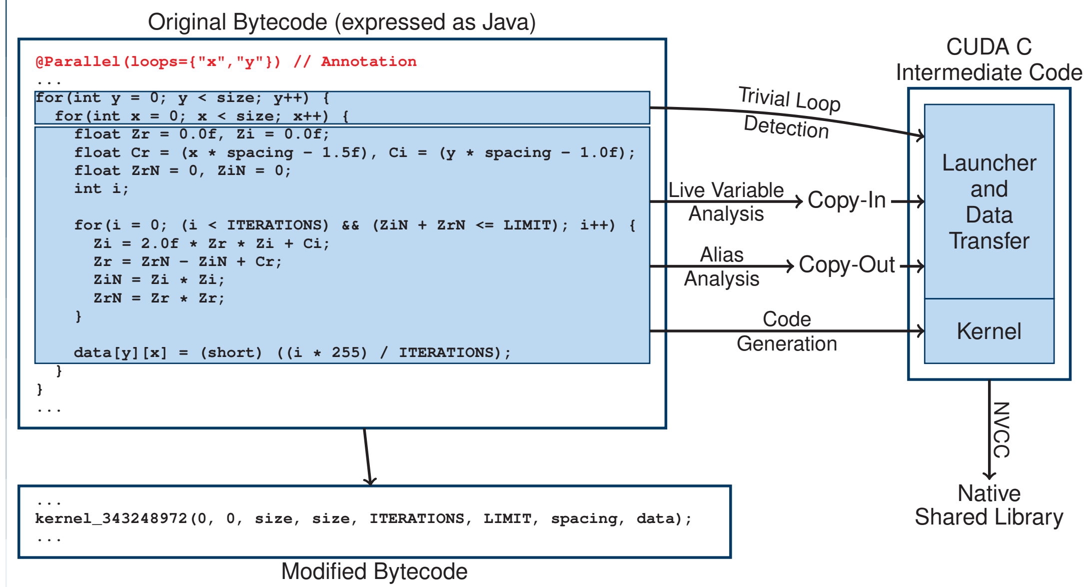
Offloading Java to Graphics Processors

Pete Calvert (prc33@cam.ac.uk)

Contribution

This work allows compilation of high-level Java bytecode for CUDA graphics processors, with automated data copying. It achieves significant speedups that are similar to non-automated work.

Overall Process



Trivial Loops

Defined as *natural loops* with a single loop exit comparing an *increment variable* with an expression:

```
while(i < LIMIT) {...i++; ...i += 3; ...}
```

Detected using *dominator analysis* and dataflow analysis to determine increment variables.

Parallelisable loops are marked with *Java annotations* using variable names (debugging information must therefore be provided).

Kernel Parameters

- Parameters must be eagerly copied to the GPU.
- *Copy-In* parameters are determined by live variable analysis on the kernel body.
- *Copy-Out* parameters are determined by alias analysis, falling back to all *copy-in* parameters if the analysis fails to converge.

Code Generation

CUDA C code is generated from an internal representation in which the operand stack is replaced by a dataflow graph. This avoids stack manipulation in the C code.

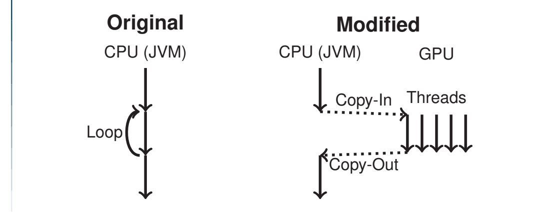
Supported:

- All primitive data types and multi-dimensional arrays of these.
- Method calls (excluding recursion due to hardware limits).
- All arithmetic and some of `java.lang.Math`.
- Basic use of objects.

Unsupported:

- Exceptions.
- Synchronisation Monitors.
- Object inheritance.

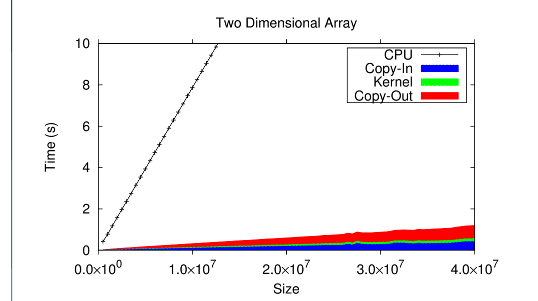
Execution



Performance Results

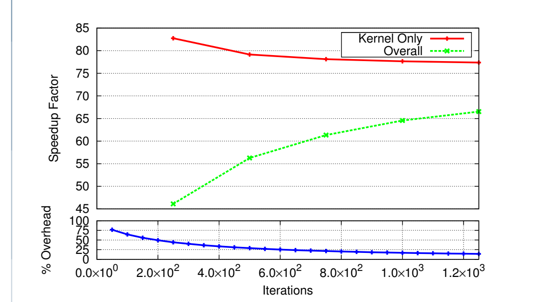
All measurements were taken using a GeForce GTX 260 (216 cores, 1.24GHz, 896Mb global memory) on a machine with two 3.2GHz Pentium 4 processors.

Simple Numeric Test *(double precision sine and cosine)*:



Transferring large numbers of objects incurs significant overheads due to the number of JNI calls, and ensuring an object is not imported twice.

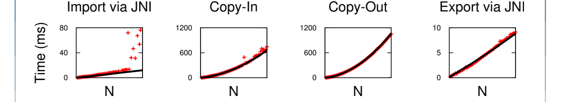
Mandelbrot Set Computation:



Java Grande Benchmark Suite: Performance achieved for both the *Fourier Series* (speedup of 187×) and *IDEA Encryption* (8.7×) benchmarks were broadly comparable to those claimed by JCUDA using custom CUDA code.

Data Transfer Overhead Prediction

Parameters for an overhead model were measured using the *numeric* benchmarks, and were then used to accurately predict the overheads in other benchmarks covering 1D and 2D cases. The fit for the *Mandelbrot* benchmark is shown below:



This model could be employed to judge whether offloading computation to a GPU would be worthwhile.

Conclusion and Future Work

This work has shown that GPU code development can be eased whilst maintaining performance.

'Multi-Pass Loops' (as in Leung) would avoid state moving back and forth from the GPU in iterative algorithms (such as *Game of Life*).

Multiple GPUs: Either modification flags (adding runtime overheads), or further static analysis is needed to determine the data affected by each card.

Reduced Copies: Data is currently copied at the granularity of whole variables. This can be excessive for large arrays and objects. Further work might look at how these copies can be restricted.

Related Work

'Automatic parallelization for graphics processing units' (*A. Leung, O. Lhoták, G. Lashari.* In *PPPJ 2009*): Entirely automatic approach within the Jikes research virtual machine—not possible to give user hints.

JCUDA (*Y. Yan, M. Grossman, V. Sarkar.* In *Euro-Par 2009*): Integrates CUDA kernels into Java through extra syntax—kernels still written in C.

Acknowledgements

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

The full dissertation is available from `http://people.pwf.cam.ac.uk/prc33/`.