MONASH University
Engineering

Department of Electrical and
Computer Systems Engineering

ECE4095 Final Year Project 2016

James Anastasiou
Supervisor: Dr. David Boland

Your Code Sucks: Should you pay someone to fix it?

Why Your Code Sucks

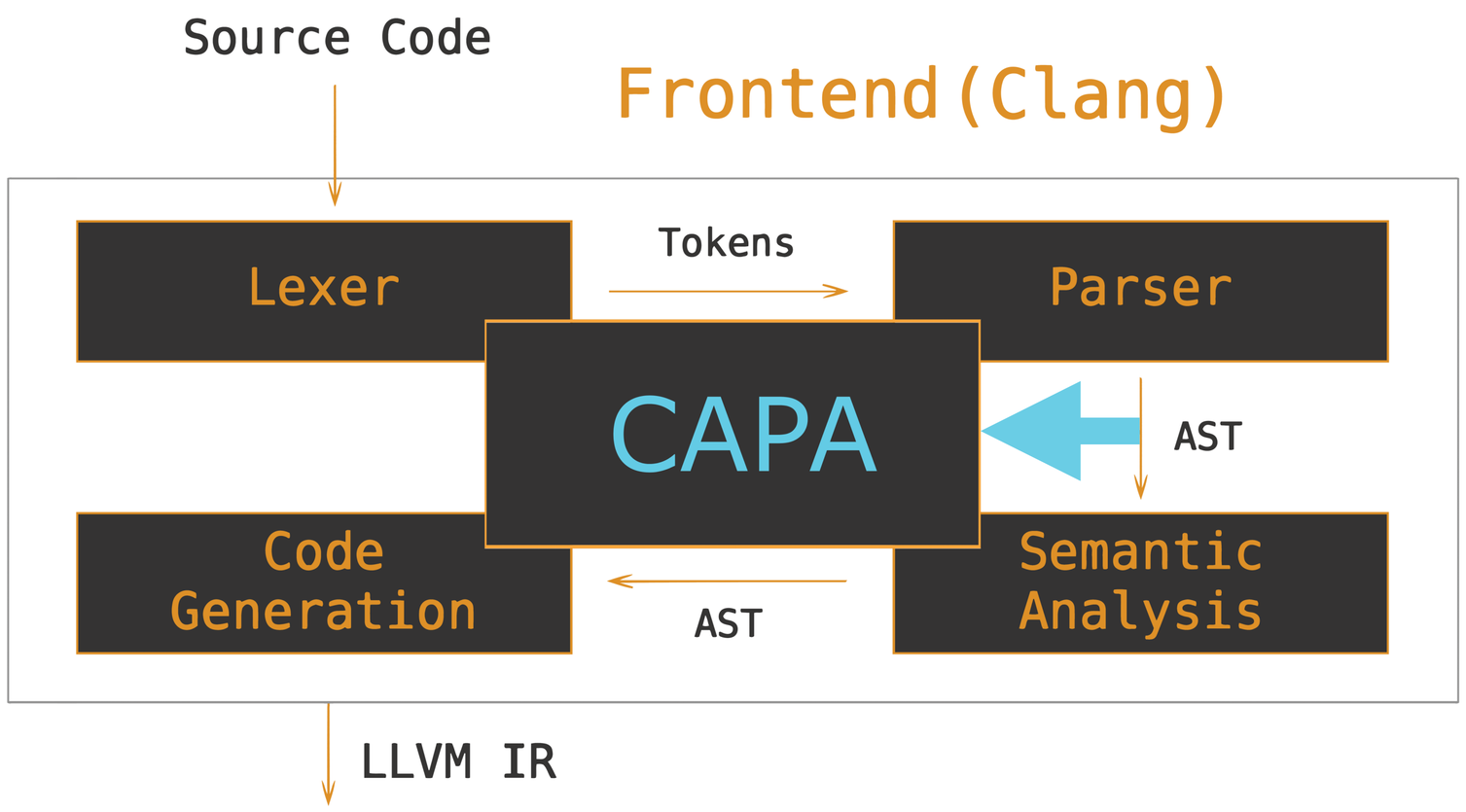
You're unlikely to have written your program in such a way that it exploits parallelism to extract performance. Modern compilers try to do this for you, however a targeted approach provides superior results. What if you had a tool that could tell you what to fix?

Project Aim

To develop a source code analyser which generates a report detailing how regions of a codebase could be potentially parallelised, exploiting the massively parallel computational power of Graphics Processing Units (GPUs) through General Purpose GPU programming.

Enter CAPA

CAPA is a tool which utilises static analysis techniques to identify and isolate potentially parallelisable regions of code, and generate a report detailing potential performance improvements that could be gained by refactoring the identified code onto a GPU.



Hooking Into Clang

CAPA hooks into the Clang C Compiler to perform Static Analysis on your source code. Clang takes care of lexing and parsing the source file, generating an AST at which point CAPA hooks into Clang via the libtooling interface. CAPA leaves the heavy lifting to Clang, and focuses entirely on finding optimisation opportunities in your code.

CAPA exploits Clangs C level optimisation engine as well, utilising the Constant Folding to extract as much information from the source as possible.

Because Clang is capable of compiling many C family languages, with some additional work CAPA could be used to analyse more than just C, working with C++, Obj-C and Obj-C++

What CAPA Does

CAPA traverses the Abstract Syntax Tree generated by Clang during compilation, searching for patterns within the representation that identify operations which can be performed significantly faster on a GPU.

CAPA specifically looks for Map, Reduce and Scan operations which form the parallel primitive operation set. It also looks for Matrix Multiplication which has been highly optimised for performance on a GPU. Other subroutines handling linear algebra can also be found.

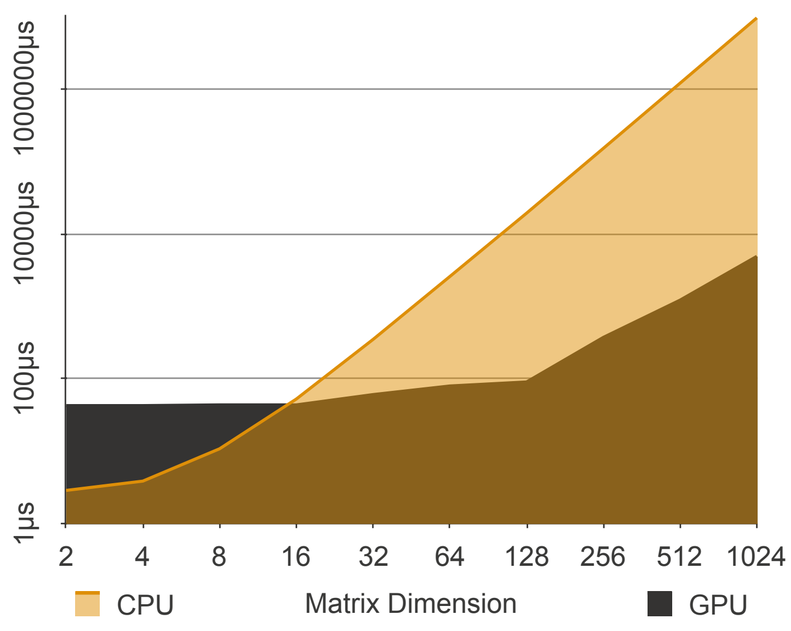
CAPA generates a report detailing how the analysed code can be optimised, providing information about the expected performance benefits refactoring could provide.

```
01.  #include ...
02.
03.  int main(){
04.      const size_t ELEMS = 1000*1000;
05.      float starting_vec[ELEMS];
06.      initialise(starting_vec, ELEMS);
07.
08.      for (size_t i = 0; i < ELEMS; ++i){
09.          starting_vec[i] /= 2;
10.          starting_vec[i] += 4;
11.      }
12.
13.      // Calculate mean
14.      float k = 0;
15.      for (size_t i = 0; i < ELEMS; ++i)
16.          k += starting_vec[i]/ELEMS;
17.
18.      // Renormalise all values and compute cumulative sum
19.      float cum_sum[ELEMS];
20.      cum_sum[0] = starting_vec[0]/k;
21.      for (size_t i = 1; i < ELEMS; i++)
22.          cum_sum[i] = starting_vec[i]/ELEMS + cum_sum[i-1];
23.      /* ... */
24.  }
25.
26.  void mmult(float **A, float **B, float **C, size_t dim){
27.      for (size_t i = 0; i < dim; ++i)
28.          for (size_t j = 0; j < dim; ++j)
29.              for (size_t k = 0; k < dim; ++i)
30.                  C[i][j] += A[i][k] * B[k][j];
31.  }
```

```
CAPA Report

Summary: TotalFiles=1 Files With Improvements=1

/home/james/Projects/CAPA/case/CodeUnderTest.cpp:26:1
Pattern: Vectorisable Function Declaration Priority: 3 Info: Function Declaration
void mmult(float ** A, float ** B, float ** C, size_t dim);

/home/james/Projects/CAPA/case/CodeUnderTest.cpp:8:5
Pattern: Map Priority: 3 Info: Stride Size: 1. Number of Elements: 1000000.
Potential Speedup: 158.03 ~ 140.51
for (size_t i = 0; i < ELEMS; ++i){
        starting_vec[i] /= 2;
        starting_vec[i] += 4;
    }

/home/james/Projects/CAPA/case/CodeUnderTest.cpp:15:5
Pattern: Reduce Priority: 2 Info: Stride Size: 1. Number of Elements: 1000000.
Potential Speedup: 30.30 ~ 34.33
for (size_t i = 0; i < ELEMS; ++i)
        k += starting_vec[i]/ELEMS

/home/james/Projects/CAPA/case/CodeUnderTest.cpp:21:5
Pattern: Scan Priority: 2 Info: Stride Size: 1. Number of Elements: 1000000.
Potential Speedup: 19.72 ~ 30.11
for (size_t i = 1; i < ELEMS; i++)
        cum_sum[i] = starting_vec[i]/ELEMS + cum_sum[i-1]

/home/james/Projects/CAPA/case/CodeUnderTest.cpp:27:5
Pattern: Matrix Multiplication Priority: 1 Info: A Matrix Multiply
Potential Speedup: 16.72 ~ 229.00
for (size_t i = 0; i < dim; ++i)
        for (size_t j = 0; j < dim; ++j)
            for (size_t k = 0; k < dim; ++k)
                C[i][j] += A[i][k] * B[k][j]


[CAPA v0.10.2]
```

What's The Point?

GPUs can perform many computationally expensive operations orders of magnitude quicker than their CPU competitors. Take a single precision matrix multiplication, including all memory operations overhead a trivial implementation on a GPU can perform over 200x faster than the serial version. Most programmers do not consider GPGPU programming as a solution, even though it could be a superior option. CAPA seeks to provide developers with an automated tool that helps them identify whether it is worth employing a specialist to optimise the codebase.



A Little Bit Extra

Currently I am looking at expanding CAPA into providing recommendations for FPGA development. Extending the same analysis techniques into finding and

Where To Find It

Source for CAPA can be found at:
http://github.com/jhana1/CAPA

Source for CAPA-Benchmark can be found:
http://github.com/jhana1/CAPA-Benchmarker