EigenCFA: A CFA for the λ-Calculus on a GPU

Tarun Prabhu, Shreyas Ramalingam

Abstract

We propose a novel technique to perform 0CFA for a 2-ary CPS λ-calculus on a GPU. By carefully encoding the program and using matrix operations exclusively to perform the abstract interpretation, we believe that we can improve on existing control-flow analyses.

1 Team Members

- Tarun Prabhu: is interested in compiler design and static analysis.
- Shreyas Ramalingam: is interested in optimization for scientific computing.

2 Introduction

Traditional zero-order control flow analysis (0CFA) algorithms run in cubic time and there is strong evidence to suggest that it cannot be improved [1]. In light of this, parallelizing the CFA would be the obvious approach to obtain faster analyses. We propose a novel technique to perform 0CFA for a 2-ary continuation passing style (CPS) λ-calculus on a GPU. Since GPU's are very good at performing matrix multiplications efficiently, all the store lookups and updates will be performed using matrix multiplication.

2.1 The Language

$$\begin{aligned} call &\in Call ::= (f\ e_1\ e_2) \\ f, e &\in Exp = Var + Lam \\ v &\in Var \text{ is a set of identifiers} \\ lam &\in Lam ::= (\lambda(v_1\ v_2)\ call) \end{aligned}$$

2.2 Abstract State Space

$$\begin{aligned} \hat{\varsigma} \in \Sigma &= Call \times \widehat{Store} \\ \hat{\sigma} \in \widehat{Store} &= Var \rightarrow \widehat{Lams} \\ \hat{L} \in \widehat{Lams} &= \mathcal{P}(Lam) \end{aligned}$$

2.3 0CFA Abstract Semantics

As a transition relation $(\Rightarrow) \subset \Sigma \times \Sigma$

$$\begin{array}{rcl}
(\llbracket(f\ e_1\ e_2)\rrbracket, \hat{\sigma} & \rightsquigarrow & (call, \hat{\sigma}) \\
(\llbracket(\lambda(v_1\ v_2)call)\rrbracket & \in & \hat{\mathcal{E}}(f, \hat{\sigma}) \\
\hat{L_1} & = & (\hat{E})(e_1, \hat{\sigma}) \\
\hat{L_2} & = & (\hat{E})(e_2, \hat{\sigma}) \\
\hat{\sigma}' & = & \hat{\sigma} \sqcup [v_1 \mapsto \hat{L_1}, v_2 \mapsto \hat{L_2}
\end{array}$$

where,

$$\begin{array}{rcl}
\hat{\mathcal{E}} : Var \times \widehat{Store} & \rightarrow & \mathcal{P}(Lam) \\
\hat{\mathcal{E}}(v, \hat{\sigma}) & = & \hat{\sigma}(v) \\
\hat{\mathcal{E}}(lam, \hat{\sigma}) & = & \{lam\}
\end{array}$$

2.4 Helper Functions

A number of helper functions will be used to lookup information about lambda expressions and call sites. These functions will return vectors which will be used to lookup variables and lambda expressions in the store. A brief description of the functions is given below:

$$\begin{array}{rcll}
\textbf{Fun} & : & \vec{Call} \rightarrow \vec{Exp} & (\textit{extracts the function expression from a call site}) \\
\textbf{Arg}_1 & : & \vec{Call} \rightarrow \vec{Exp} & (\textit{extracts the first argument expression from a call site}) \\
\textbf{Arg}_2 & : & \vec{Call} \rightarrow \vec{Exp} & (\textit{extracts the second argument expression from a call site}) \\
\textbf{LamCall} & : & \vec{Lam} \rightarrow \vec{Call} & (\textit{extracts the call site from a lambda term}) \\
\textbf{LamVar}_1 & : & \vec{Lam} \rightarrow \vec{Exp} & (\textit{extracts the first formal argument from a lambda term}) \\
\textbf{LamVar}_2 & : & \vec{Lam} \rightarrow \vec{Exp} & (\textit{extracts the second formal argument from a lambda term})
\end{array}$$

2.5 Pseudo-code

1. Pre-process program and populate datastructures which will be used by the helper functions described in 2.4.

2. Initialize the store and copy it into global memory of the GPU.

3. **foreach** *call* **in** *program*

$$\begin{array}{rcl}
\vec{L} & = & \vec{\sigma}(\textbf{Fun}(call)) \\
\vec{L_1} & = & \vec{\sigma}(\textbf{Arg}_1(call)) \\
\vec{L_2} & = & \vec{\sigma}(\textbf{Arg}_2(call)) \\
\vec{call'} & = & \vec{\sigma}(\textbf{LamCall}(\vec{L})) \\
\vec{v_1} & = & \vec{\sigma}(\textbf{LamVar}_1(\vec{L})) \\
\vec{v_2} & = & \vec{\sigma}(\textbf{LamVar}_2(\vec{L})) \\
\sigma\prime & = & \sigma + \vec{L_1} \times \vec{v_1^T} + \vec{L_2} \times \vec{v_2^T} \\
 & : & \textit{Here } \sigma(\bullet) \textit{ is a lookup into the store}
\end{array}$$

4. End

3 Suitability for GPU acceleration

Encoding the entire problem as a series of operations on matrices, we will utilize the inherent power of a GPU to perform matrix operations efficiently. Each iteration in our scheme has at least 2 matrix multiplications and 1 matrix addition. While this approach to doing 0CFA is not something which is usually implemented on a CPU, we believe that for large problem sizes, the GPU method will outperform current CPU techniques in terms of speed.

3.1 Amdahl's Law

It is difficult to quantify the speed-up which this technique will provide, primarily because it doesn't seem likely that this technique will be used on a CPU to perform an analysis. If, for the sake of argument, we do assume that this exact method will be implemented on a CPU, the speedup will be proportional to the speedup that is obtained for matrix multiply on the GPU.

3.2 Synchronization and Communication

In the initial implementation, the CUBLAS library will be used to perform the matrix multiplication. The interpretation will be controlled from the host with several kernel calls being made to perform the lookups and updates (which are matrix operations). In order to improve the space and time efficiency, a matrix multiplication library could be developed which would operate on bit vectors. Another improvement would be to have more of the CFA computation on the kernel instead of the host.

3.3 Copy Overhead

For every call site being evaluated, we would need to copy the values of L_1, L_2, v_1, v_2 from the host to the device memory. This could become very signficant unless it is optimized way.

4 Challenges

We foresee a number of challenges in implementing our scheme:

4.1 Kernel Call Overhead

Currently, at least 3 kernel calls will be made per call-site. The overhead involved in setting up these kernel calls could become significant. A major challenge would be to perform as many lookup and update operations as possible within a single kernel call. This would entail writing a very customized matrix multiplication library tailored specifically to the current task. This would also reduce the overhead of copying vectors to the device at every call-site.

4.2 Memory Requirements

A naive encoding of the program into matrices would result in very large, very sparse matrices. Fitting such large matrices into the limited memory of the GPU is a major challenge. Encoding the matrix as a bit vector would lead to big gains on this front, at the expense of making the multiplication operation more complex.

4.3 Helper Functions

Efficient helper functions would significantly improve performance since there are 6 calls being made to these functions per call-site. The most efficient design for these functions has yet to be determined.

5 Implementation Details

Parsing lambda calculus is a task more suited to functional programming languages like Scheme. We propose to use a pre-processor written in PLT Scheme which will generate the lookup tables and the initial store. The main CFA engine will be written in CUDA and will first read in the data generated by the pre-processor and then proceed with the analysis.

References

[1] David van Horn, Harry G. Mairson *Flow analysis, linearity, and PTIME*. Static Analysis Symposium (SAS) 2008.