Eliminating Buffer Overflows in MOOC Labs by Visualizing Memory Accesses in Parallel Programs

Abdul Dakkak
University of Illinois at
Urbana-Champaign
dakkak@illinois.edu

ABSTRACT

The recent popularity of Massive Open On-line Courses (MOOC) creates both opportunities and challenges. Since while the courses do reach thousands of students, systems and processes designed to deal with hundreds fail to scale. In the "Hetrogenous Parallel Programming" class offered through Coursera, the main issue in scaling was system reliability due to memory indexing error encountered while running students' code. Since students' code is run on Graphical Processing Units (GPUs), the system would crash when executing certain memory errors. Aside from redundancy to avoid single points of failure, giving students' tools to visualize how their code indexes memory would not only decrease the amount of faulty code executing on the GPU, but also allow students' to visually understand code. In this project we developed Jaster: a source-to-source compiler that takes CUDA (a GPU programming language) as input and produces JavaScript code. Unlike CUDA code, which requires specialized GPU hardware, the JavaScript code is executed in the student's browser. During the JavaScript code execution, a visualization of the memory region is displayed along with it's access pattern. The tool will be deployed in the course's next installment in January 2015.

INTRODUCTION

MOTIVATION

Due to the fact we think sequentially, parallel programming is hard to get right. A slew of other problems crop up due to the indeterministic nature of parallel code. Logic errors such as race conditions, deadlocks, and starvation cause indeterminisiticly generate unexpected outcome or cause system errors. Since parallel languages are also hosted within a low level language, you inherit aditional problems such as: unsafe type casts, buffer overflows, bad array index calculations.

Current CUDA debugging tools such as Nsight for Visual Studio or `cuda-gdb` for Unix operate by suspending the GPU. This means that one needs at least two GPU cards are needed while debugging GPU code. Futhremore, these debugging tools are quite involved — requiring the learning of new commands, understand a different workflow, and because these tools are proprietery cannot not be easily integrated with the course's system.

Background

DATASET

Code Similarity

Since template code was provided to the students, and the code follows the course lectures, structural similarity within the solutions should be common. To hone in on the visualization, we mine the dataset (which contains a sampling of 100 programs from each lab [1]). Similarity between two codes is measured using a modified version of the Zhang-Shasha tree edit distance algorithm.

The Zhang-Shasha algorithm measure the distance between two trees with respenct to distance functions. The distance functions decide the weight of inserting, deleting, and renaming nodes in the AST tree.

Cannonicalization of the AST Representation

Because of the high complexity, we prune the tree to only include the CUDA kernel part. Since two codes might have different AST representation but same semantic meaning, we perform a normalization step on the AST before computing the distance. We first replace all the identifiers with a canonical name. Identifiers that are not within the body of the function are deleted, since they are given as part of the template code. Identifiers within the function body are computed based on a hash of how the identifier is assigned to within the code. Literals (such as integers) are ignored since they do not contribute to any structural difference in the code (they do contribute to the hash calculation for the identifiers however). `for` and `do` while loops are rewritten to their `while` representation, conditional expressions (expressions of the form `var = cond ? then : else`) are rewritten to `if (cond) { var = then; } else { var = else; }`, `if` conditions are rewritten so that the biggest node is within the `then` branch of the condition. Finally, for compound expressions are canonicallize the code based on the following rewrite rules:

- Types of variable declarations are removed : statements such as `int x = 4;` are replaced wth `x = 4;`
- Non-side effecting statements are pruned : statements are `x;` or `f();` where `f` is known to be a pure function get pruned
- Compound binary expressions are reordered based on operator precidence: $a\,\mathtt{op}_1\,(b\,\mathtt{op}_2\,c) \rightarrow (b\,\mathtt{op}_2\,c)\,\mathtt{op}_1\,a$ iff $\mathtt{op}_2$ has lower precidence than $\mathtt{op}_1$
- Leaf binary operations are reordered based on the lexographical ordering of the leafs : $a\,\mathtt{op}\,b \rightarrow b\,\mathtt{op}\,a$ iff b is lexographically less than a

[1] The database contains over 2 million programs, but since the code similarity algorithm is computationally expensive ($O(n^4)$ in time and $O(n^2)$ in space), we were not able to perform full correlation between the codes

- Strength reduction is applied : expressions such as `x * 2` get replaced by `x << 1`.

- Constant propagation is applied : expressions such as `a = 3; b = a * 4;` get replaced by `a = 3; b = 12;`

To lexographically order the AST leaf node we use the follwing heuristic: literals of the same type use the $<$ operation for ordering, literals of different types are first ordered based on their value and then, in case of a tie, based on the byte-count of their type, literal nodes are always lexographically less than identifiers, and identifiers use string lexographical ordering.

In cases where a compound node cannot be canonicallized, we mark the node for special processing during the distance calculation. During computation, the edit distance of these nodes is computed as the minimum distance of its children.

When computing the edit distance, the cost for replacement of an identifier is proportional to the Levenshtein distance of strings. Deletion and insersions contribute 1 to the cost.

Zhang-Shasha Tree Edit Distance

Memory Access Similarity

BACKEND IMPLEMENTATION

The implementation is divided into two parts.

C++ Parser

The first is a C++ library the uses CLANG RecursiveAST interface to parse CUDA code and storing the Abstract Syntax Tree (AST) nodes in a mutable representation We developed a new AST interface, since the one provided by CLANG is immutable. Our AST interface stores not only the node information, but also the location of the code within the input file, the text underlying the node, and parent child relashionship. Once the parsing is complete, our datastructure dumps the information into JSON form. A C++ expression such as `x + 1` generates the following JSON output:

```
{
  type: "BinaryExpression",
  operator: "*",
  left: {
    type: "Identifier",
    name: "x",
    loc: {
      start: { line: 2, column: 8 },
      end:   { line: 2, column: 9 }
    }
  },
  right: {
    type: "Literal",
    value: 1,
    loc: {
      start: { line: 2, column: 10 },
      end:   { line: 2, column: 11 }
    }
  }
}
```

We use the Mozzila AST specification, to name the fields within the JSON datastructure. While not a standard, it is the only documented JavaScript AST serialization specification that we are aware of. Futhermore, tools to read, verify, and traverse the AST in the Mozilla Specification exist.

C++ Parser

JavaScript Transpiler

Since JavaScript does not contain type information, and types are needed to properly maintain the semantics of C++ — such as truncation of floats when assigned to integers or offset calculation for array indexing. The transpiler adds type information in the `functionStack$` type field. Code such as `int x = 1.2;` results in the following JavaScript code

```
functionStack$['x'] = Math.floor(1.2);
lib.setType(functionStack$, 'x', {
  type: 'TypeExpression',
  addressSpace: [''],
  qualifiers: [],
  bases: ['int']
});
```

The type information is queried when calculating the offset in the heap when indexing an array. For example, for code such as

```
int * y;
double * x;
....
x[2] = 5 + y[1];
```

then the element size of the array `y` is `sizeof(int)` $= 4$. So, `y[1]` causes 4 bytes to be read from index 4 of `y` and interpreted as an integer. Similarly, `sizeof(double)` $= 8$ bytes must be set at index $2 *$ `sizeof(double)` $= 16$ from array `x` as a double precision number.

RUNTIME

C++ Types

Unlike

The C++ Heap

CUDA Execution

Avoiding Blocking

Maintaining the SourceMap

FRONTEND IMPLEMENTATION

RESULTS

LIMITATIONS

FUTURE WORK

REFERENCES

1. Adobe Acrobat Reader 7. **http://www.adobe.com/products/acrobat/.**

2. Anderson, R. E. Social Impacts of Computing: Codes of Professional Ethics. *Social Science Computer Review December 10*, 4 (1992), 453–469.

3. How to Classify Works Using ACM's Computing Classification System. **http://www.acm.org/class/how_to_use.html**.

4. Klemmer, S. R., Thomsen, M., Phelps-Goodman, E., Lee, R., and Landay, J. A. Where do web sites come from?: capturing and interacting with design history. In *Proc. CHI 2002*, ACM Press (2002), 1–8.

5. Mather, B. D. Making up titles for conference papers. In *Ext. Abstracts CHI 2000*, ACM Press (2000), 1–2.

6. Schwartz, M. *Guidelines for Bias-Free Writing*. Indiana University Press, 1995.

7. Zellweger, P. T., Bouvin, N. O., Jehøj, H., and Mackinlay, J. D. Fluid annotations in an open world. In *Proc. Hypertext 2001*, ACM Press (2001), 9–18.