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 "Heterogeneous 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. Since CUDA memory accesses are linearized, the tool makes use of source code classification to reconstruct the un-linearized access. The tool will be deployed in the course's next installment in January 2015.

INTRODUCTION

With the expansion of the reach of programming courses through Massive Open Online Courses, educators are rethinking the way these courses are offered. The expectation that a student would setup a development environment (setting a compiler, learning new tools for debugging, and setting up an IDE) would limit the reach of the course, since it would exclude people who cannot install such software or cannot afford it. Furthermore, it requires at least a week for students to orient themselves with the new flow, which is long for the relatively short courses. Thus multiple courses now offer a way to how on the problem through web interfaces.

The trend is not limited to educators. Microsoft [12] and Eclipse [9] both have an online version of their IDE and compiler. Similarly, most languages now offer a playground area where users can post and evaluate code.

All of these systems assume that you are running generic code, but what if you'd like to teach a course that requires special hardware. Furthermore, the hardware cannot be emulated. This was the challenge faced when we setup the Coursera Heterogeneous Parallel Programming course. We would like users to have as much control while still maintaining availability and fairness[1].

Based on user's surveys the most common complain is the limited number of developer tools offered through the site. In this project we develop a tool to visualize how a CUDA program executes and how memory is addressed. We first write a translator from CUDA to JavaScript that maintains the original semantics. We classify labs from previous iterations of the course, which contain the most common techniques for CUDA execution and memory addressing. Then, given a buggy piece of code we find the closest lab to not only determine how to layout the visualization, but also determine the un-linearlized memory access. An overview of our pipeline is show in figure 3.

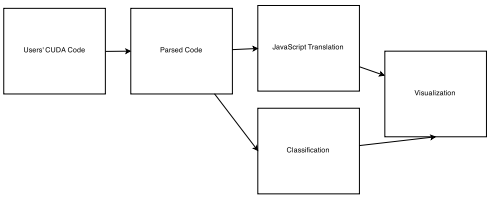

Figure 1. The pipeline of our project. In this flow code is first parsed, transcompiled, classified, and then visualized.

Previous Work

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 intermittently generate unexpected outcome or cause system errors. Since parallel languages are also hosted within a low level language, you inherit additional problems such as: unsafe type casts, buffer overflows, bad array index calculations. The variety of these bugs makes courses such as HPP not approachable for students who do not persevere for the first few weeks.

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. Furthermore, these debugging tools are quite involved — requiring the learning of

[1]The system is live today at `webgpu.hwu.crhc.illinois.edu`

new commands, understand a different work flow, and because these tools are proprietary cannot not be easily integrated with the course's system.

Other than low level debugging tools, we are not aware of any work on to visualize CUDA program execution — and certainly not one that does so by transpiling CUDA to JavaScript.

C++ PARSER

The first is a C++ library the uses CLANG RecursiveAST [6] 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 relationship. Once the parsing is complete, our data structure 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 Mozilla AST specification, to name the fields within the JSON data structure. While not a standard, it is the only documented JavaScript AST serialization specification that we are aware of. Furthermore, tools to read, verify, and traverse the AST in the Mozilla Specification exist.

Even though the input source code might not be parse-able as CUDA code without macro expansion, our tool performs the macro expansion but generates the source mapping to the original unprocessed text.

JAVASCRIPT TRANSPILER

The JavaScript source-to-source compiler can be divided into multiple steps. First, code must maintain the same type meaning — even though JavaScript is dynamically typed, second it must perform, finally it must contain enough meta data to backtrack to the original input program.

Pointer Types in JavaScript

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

RUNTIME

Several design decisions in the runtime were influenced by the limitations of current JavaScript engines [11, 13, 15]. We try to make use as much of the newer JavaScript features as possible that are able to achieve good performance, while still maintain shims to allow us to support older browsers. The runtime therefore makes heavy use of WebWorkers [2], TypedArrays [8, 3], promises [5, 1], and generators in our implementation.

C++ Types

JavaScript contains only one literal numeric type: an IEEE double precision double. To emulate the other types in JavaScript efferently we again use TypedArrays since they are able to represent 8, 16, and 32 bit signed and unsigned integers. Scalar C++ operations on these types get translated as operations on a 1 element array of the corresponding type. To emulate 64 bit integers, we use two 32 bit integers and perform the high and low bit calculation [14]. We experimented with other ways of emulating 64bit integers, but found using 32bits was the most efficient.

The C++ Heap

We use TypedArrays to model the C++ Heap. This is now a common technique when transpiling from unmanaged languages. At the initialization of the program, then runtime allocates a slab of memory. This memory is then used as a pool where allocations are honored by taking a slab of the buffer.

Since CUDA memory is in a different address space than the host memory, the runtime creates two heaps. Errors are thrown if operations done on the allocated memory result in undefined behavior (operating on GPU memory using host memory functions for example).

A typical example is a `cudaMalloc` call which takes the form of

```
cudaMalloc((void**)&deviceB,
    numBRows*numBColumns*sizeof(float))
```

The generated JavaScript code generated not only allocates the memory on the GPU memory heap and marks it, but also steps the passed in reference to the newly allocated buffer.

```
lib.cuda.cudaMalloc(
state$,
functionStack$,
lib.ref(state$, functionStack$, 'deviceC'),
  functionStack$['numARows']*
  functionStack$['numBColumns']*
  lib.c.sizeof(state$, 'float'),
 [
   'deviceC',
   'numARows',
   'numBColumns'
 ])
```

The last argument to `cudaMalloc` is used by the classifier to determine whether the allocated memory is one-, two-, or three-dimensional. The labels also are used for indexing.

CUDA Execution

We use a combination of asynchronous operations and true multi threading to simulate CUDA execution. The code kernel invocations are dispatched into a thread pool. Each thread pool then invokes a collection in a round robin way generating a promise. Only when the promises are fulfilled between all the threads can program continue execution. By performing this technique we do not perform any blocking operation on the main UI thread.

Since JavaScript threads cannot share data, we develop a communication language between threads and the master thread (which has the data). A thread requesting data from global memory generates a message which the master thread corresponds to. Effectively we translate the shared threaded execution model to a message passing actor model.

Performance

When trying to emulate the execution of GPU code, one must observe that GPUs are well suited for the types of labs in the HPP MOOC and that CUDA is a low level programming language. Given that JavaScript is dynamically typed, we make use of a lot of performance opportunities afforded to us by the standard. We use TypedArrays [4, 7] to emulate the heap, for example, while webworkers are used to make use of multiple threads of execution.

By creating artificial boundaries within functions, we are able to schedule the code to run on the main thread while maintaining the UI's responsiveness. Furthermore, we store all stack information in a special variable which allows us pause and resume the execution frame at any point.

To maintain the same execution time as CUDA, we rescale the inputs to be a tenth of the size of data that is traditionally given for lab execution.

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 picked from compilable programs in a from over 2 million program database [2].

We use the Zhang-Shasha [16] algorithm measure the distance between two trees with respect to distance functions. The algorithm complexity is $O(n^4)$ in time and $O(n^2)$ in space, but that's mitigated by the fact that this computation is done on the client's machine. Zhang-Shasha's algorithm determines the distance between two pairs of ASTs by calculating the number of insertions, deletions, and renaming of the nodes required to equate the AST nodes. Since two noes might be different in the AST but semantically the same, we perform canonicalization to mimic semantic equivalence. Since we still want to be able to present the user with the AST differences, we maintain the original node information while performing the canonicalization.

Canonicalization 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 canonicalize the code based on the following rewrite rules:

- Types of variable declarations are removed : statements such as `int x = 4;` are replaced with `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 precedence: $a\,\mathtt{op}_1(b\,\mathtt{op}_2\,c) \rightarrow (b\,\mathtt{op}_2\,c)\,\mathtt{op}_1\,a$ iff $\mathtt{op}_2$ has lower precedence than $\mathtt{op}_1$
- Leaf binary operations are reordered based on the lexicographical ordering of the leafs : $a\,\mathtt{op}\,b \rightarrow b\,\mathtt{op}\,a$ iff b is lexicographically less than a

[2]We are limited to a small set of programs because of the computational complexity of the algorithm. In the coming weeks we plan on we plan on performing full correlation between the codes. since the code similarity algorithm is computationally expensive

- Strength reduction is applied : expressions such as `x * 2` get replaced by `x << 1` or `x * 1 = x`.
- Constant propagation is applied : expressions such as `a = 3; b = a * 4;` get replaced by `a = 3; b = 12;`

To lexicographically order the AST leaf node we use the following 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, the tie is broken by considering the bytesize of the types, literal nodes are always lexicographically less than identifiers, and identifiers use string lexicographical ordering.

In cases where a compound node cannot be canonicalized, 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 insertions contribute 1 to the cost.

Classification in the Pipeline

Since C code has no multidimensional arrays, all multidimensional arrays are linearized as one dimensional ones — an access such as `x[ii * with + jj]` is a linearized form of `x[ii][jj]`. This posses a problem when we try visualize the accesses. To mitigate that, we use both canonicalization as well as finding the nearest source code with known access pattern to convert the linearized access into its higher order form.

FRONTEND IMPLEMENTATION

We use React [10] to implement our visualization. React constructs a virtual DOM and only performs DOM updates when the tree is different enough from the one being displayed. By having batched rendering, versus fragmented ones, we were able to achieve good performance over a pure SVG based rendering.

Furthermore, by having each thread be a react component, our code is very simple to reason about. React also provides good binding between the data and visual elements. Changing the data in one of it's components triggers a rerendering of that component. If a component has children, then all of the children may get rerendered as well.

React also contains Mixins to memoize some rendering. If a component is requested to be redrawn, then React can ignore that request if it can show that the component data did not mutate.

LIMITATIONS AND FUTURE WORK

The main limitation of the project, from a production point of view, is the excessive use of modern JavaScript features that are not available in older browser releases. More time would be spent providing shims for the missing features.

There is no reason for one not to be able to zoom into the execution grid, pause the threads, examine the stack, etc... due to limited time, however, we did not have time to implement these features.

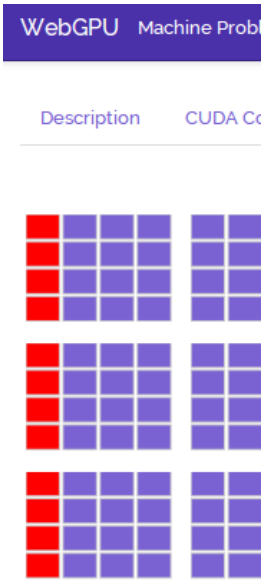

Figure 2. The execution maintains information about memory writes and is able to detect buffer overflow errors. These are highlighted in red and the user can hover over each thread (rectangle) to determine the thread that caused the bad memory access.

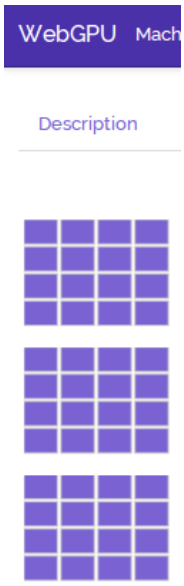

Figure 3. Execution with no memory issues produces no errors but still might give insight to the student on how they access memory — coalescing the memory accesses is a major optimization for GPUs, for example.

We believe that the tree edit distance can be optimized to allow us to perform correlation between all the programs. The algorithm is amendable to parallelism across cores and nodes in a cluster and using some cleaver caching methods as well as techniques found in the graph database community can remove a lot of the redundant computation.

Finally, we plan on releasing this tool in a beta form for the Coursera students and perform an in-depth study of where it succeeds and fails.

REFERENCES

1. Bonetta, D., Binder, W., and Pautasso, C. Tigerquoll: parallel event-based javascript. In *ACM SIGPLAN Notices*, vol. 48, ACM (2013), 251–260.
2. Green, I. *Web Workers: Multithreaded Programs in JavaScript*. " O'Reilly Media, Inc.", 2012.
3. Grimmer, M., Würthinger, T., Wöß, A., and Mössenböck, H. An efficient approach for accessing c data structures from javascript. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ACM (2014), 1.
4. Guha, A., Saftoiu, C., and Krishnamurthi, S. The essence of javascript. In *ECOOP 2010–Object-Oriented Programming*. Springer, 2010, 126–150.
5. Kambona, K., Boix, E. G., and De Meuter, W. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, ACM (2013), 3.
6. Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, IEEE (2004), 75–86.
7. Maffeis, S., Mitchell, J. C., and Taly, A. An operational semantics for javascript. In *Programming languages and systems*. Springer, 2008, 307–325.
8. Matsakis, N. D., Herman, D., and Lomov, D. Typed objects in javascript.
9. Orion, E. Eclipse orion, 2014.
10. Reynders, B., Devriese, D., and Piessens, F. Multi-tier functional reactive programming for the web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ACM (2014), 55–68.
11. Vilk, J., and Berger, E. D. Doppio: breaking the browser language barrier. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM (2014), 52.
12. Visualstudio.com. Visual studio - microsoft developer tools, 2014.
13. Vouillon, J., and Balat, V. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience* (2013).
14. Warren, H. S. *Hacker's delight*. Pearson Education, 2013.
15. Zakai, A. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ACM (2011), 301–312.
16. Zhang, K., and Shasha, D. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing 18*, 6 (1989), 1245–1262.