Project Checkpoint **Nick Roberts, Dan Cascaval**
15-418 Fall 2018 nroberts, dcascava

1 Project Information

Title	DAG: A data-parallel, graphical programming language
Website	`https://nick-and-dan.github.io/418/`
Github	`https://replace-this-link.git`

2 Progress Summary

We have identified (and iterated on) a textual syntax for the DAG language. This extends the binding syntax from our proposal and introduces parallel blocks as a method of explicitly identifying parallelism.

We have developed an abstract syntax (and type checking for it), as well as an internal tree representation. We have developed an algorithm for traversing a multi-source and multi-sink program graph and resolving all possible (high-level) translations of that graph. We attach an analysis to this to resolve nesting parallel operations into a single level (via strong inlining) and identify space requirements inherent to certain parallel translations (which might make them untenable.)

As an output of this stage, we define an internal 'abstract parallel' representation suitable to multiple back-ends, which will translate to a programmable CUDA representation (which we have also defined). We have largely resolved how to translate size information (of types and element collections) down through these representations.

We have restructured our target project: for a given set of composable DAG subroutines, we will compile a CUDA interface and implementation, which the client can access from a C++ program.

As a motivating example, our current syntax for parallel matrix multiply is as follows:

```
int dotProduct(int[] v1, int[] v2) {
  return reduce(+, 0, zipWith(*, v1, v2));
}

int[] multiplyMatrixVector(int[][] m, int[] v) {
  return parallel (m) { int[] row ->
    return dotProduct(row, v);
  };
}

int[][] multiplyMatrixMatrix(int[][] m1, int[][] m2) {
  return parallel (transpose(m2)) { int[] col ->
    return multiplyMatrixVector(m1, col);
  };
}
```

3 Refined Goals and Deliverables

We have defined a language, and we have made significant progress towards implementing a compiler for it. The nature of the project is such that a significant portion of it must be complete before we can evaluate success. In order to reliably compile even a sub-class of programs requires almost the entirety of the compiler infrastructure to be developed and tested. This still allows small modifications in the language syntax later on, and our code is modular enough to allow subsequent optimization passes to be inserted.

We have integrated certain features previously identified as 'optimizations' as key parts of the compiler. For example, kernel launch minimization is a key ingredient in allowing the language to achieve nested parallelism, which is necessary for our system of abstractions to compose. Similarly, our investment in compiler infrastructure allows some of our stretch goals to be achieved. For example: extracting parallelism via PThreads and SIMD intrinsics instead of CUDA can be done by implementing a different translation step on the conceptual parallelism in our abstract IR.

Conversely, we have decided to drop certain extra features from our initial goals. While a GUI is interesting from a usability/application perspective (and we still plan to be able to visualize DAG programs), it is not as interesting from the perspective of a 15-418 project, and as such we have re-categorized it as a 'nice to have'. Instead, to achieve good usability, we focus efforts on maintaining a concise, high-level textual syntax.

4 Poster Session Demo

For the poster session, we still plan to show a demo and visualization of a few DAG programs. We will additionally test the expressivity and performance of DAG by benchmarking it on a couple of parallel problems, and showing graphs of both performance and required development time.

5 Concerns and Issues

While we are reasonably certain of our language's translatability and applicability to a small subset of example programs, we are unsure if we can resolve the parallelism in a general program, and even less certain that this will abstract well to all parallel targets. Currently, our goal is to implement a sound translation, even if certain elements are not parallel even when it is possible that they could be.

Different targets have different constraints (for example, CUDA and memory) that could cause the program to exhibit parallelism on one target and not on another, violating our abstraction.

Additionally, we are unsure how many levels of parallelism can reasonably be translated, which affects the composability of our abstraction. We do not yet know what subset of programs representable in our source language we will reliably be able to 'fit' into our target abstraction. This is something we plan to refine iteratively, and may make language modifications in order to align more precisely.

6 Updated Schedule

- Week of 11/19
  - CUDA backend translation and indexing scheme from Abstract IR (Dan)
  - DAG Traversal translation to Abstract IR (Nick)
  - Implementing nested parallel inlining (Nick)
  - Collection sizing information analysis on AST (Dan)
  - Assess feasibility of C function integration (Both)
- Week of 11/26
  - Write various versions of circle-renderer code at multiple levels of abstraction using core language features. (Nick)
  - Develop and test benchmark programs (Both)
  - Implement constant and shared-memory optimizations as traversals on the CUDA IR, or a slightly higher level variant. (Dan)
  - Integrate C functions as map/filter arguments, if viable. (Nick)
  - Analyze performance of translated CUDA with respect to benchmark programs. (Both)
- Week of 12/03
  - Look into other CUDA optimizations (coalescing, texture memory) as passes on the existing CUDA representation. (Dan)
  - Implement additional primitives (scan, filter) in the compiler core, define optimized CUDA kernels for these. (Nick)
  - Identify and resolve performance gaps in benchmark programs (Both).
- Week of 12/10
  - Develop visualizations of chosen benchmark programs. (Dan)
  - Additional performance tuning and viable heuristics. (Nick)
  - Write final report. (Both)
  - Polish demo programs as complete (with C++) implementations. (Both)