Project Proposal
15-418 Fall 2018

Dan Cascaval, Nick Roberts
dcascava, nroberts

1 Project Information

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

2 Summary

We will design a graphical programming language for expressing data-parallel computations. Programs in this language will have the structure of a directed, acyclic graph (dag), vesting the compiler with more power and hence more responsibility. We will implement a compiler that targets CUDA and analyze the performance of the translated code over a series of benchmark programs taken from applications such as computer graphics.

A note on notation. Throughout, we use "dag" to refer to a directed, acyclic graph and "DAG" to refer to the programming language we will develop.

3 Background

In this section, we discuss approaches to expressing data parallelism in a programming language as well as the thrust of what compiling to CUDA involves.

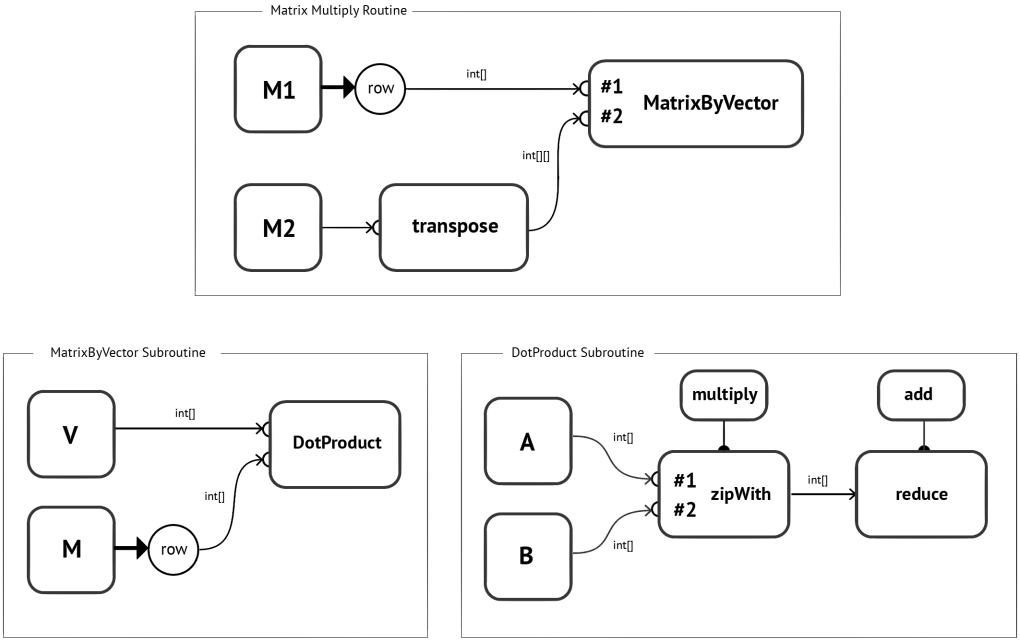
As a data-parallel language, DAG should allow the programmer to express maps, filters, and reductions over a stream of inputs. Such computations have well-understood dependency structures that allow them to be carried out in parallel. For the directed, acyclic structure of the language, we take inspiration from Haskell's list comprehensions, or more generally the usual monad structure over lists. Vertices can be understood as *terms* in the language; edges represent *binding* those terms as variables to be reused in other terms. A special edge acts as a *monadic bind* of a list term to a variable vertex which, in later computation, is taken to stand for a member of the list—this is the source of parallelism.

Concretely, take the following Haskell definition for a function that multiplies a matrix m and a vector v.

```
multiplyByVector m v = [ dotProduct row v | row ← m ]
```

The body of the definition is a list comprehension term analogous to the Python expression `[dotProduct(row, v) for row in m]`. This definition can be understood as having a DAG structure where m and v are input vertices, a monadic bind edge connects m to row, and two edges connect row and v to the term $dotProduct\ row\ v$. The monadic bind licenses the dot product computation for each row to be performed in parallel.

In terms of DAG, we provide the following implementation for a matrix multiplication routine to demonstrate basic features and semantics. Note that here, `Matrix Multiply`, `Matrix By Vector`, and `DotProduct` are all routines that can be defined by a user in DAG. `reduce`, `zipWith`, and `transpose` are all primitives that define operations that happen in parallel. These can take function arguments such as the primitives `add` and `multiply`, in this example, or any other user defined subroutine provided the types are consistent.



Targeting CUDA will involve generating a series of kernel launches using as a template appropriate map, filter, and reduce primitives. The source language provides no direct interface with CUDA, so the compiler must decide how to structure kernel launches. Relevant considerations include cache performance, memory allocation, branch divergence, fusion of kernel launches, and shared memory. Using heuristics for these considerations, the compiler will translate the declarative input program into a hopefully-efficient CUDA program, and additionally be able to indicate to the programmer where inefficiencies may arise.

4 Challenge

Translating into an efficient CUDA program involves automating decisions normally made by a human programmer with a knowledge of computer systems and parallel program performance (such as a 15-418 student). Effectively taking into account common CUDA optimization techniques such as shared memory, grouping data into blocks, and cache-friendly traversal patterns will require either some serious optimizations on part of the compiler or breaking the abstraction by exposing aspects of the target parallel platform in some understandable fashion within DAG. Achieving this balance is an open problem in API design.

We hope to discover if the same optimization techniques we applied in Assignment 2 to the circle renderer can be extended to a wider class of programs, and to what extent the compiler can automatically determine use cases and perform these optimizations on the programmer's behalf. We see this as a way for programmers to intuitively gain an understanding of program structure, quickly benefit from known advances in platform-specific performance optimizations, and simply as a method of eradicating a wide class of correctness bugs that occur when writing parallel and imperative programs in general.

5 Resources

For performing CUDA benchmarks, we will use the same machines we used in Assignment 2: the GHC machines with NVIDIA GeForce GTX 1080 GPUs. No starter code is necessary, though we may make use of a cross-platform GUI framework and various existing compilers such as NVCC or MPICC.

6 Goals and Deliverables

6.1 Target Goals

- *Define a language.* We will provide a well-defined simple, data-parallel language with a textual intermediate representation (derived from list-comprehension bindings and syntax) along with a type system that allows expression via several primitives. It is our intent that the language be compositional through mechanisms for subroutines.

- *Allow C bindings.* We aim to allow the user to define implementations of stubbed kernel functions using C, which allows versatility in the expressible computations. It is our intent that the framework define the argument signature, inside which the user can express custom logic that is applied in parallel (a kernel function for a map, or a predicate for a filter). That being said, we anticipate it will be possible to implement a large amount of desired logic using user-defined subroutines written in DAG.

- *Implement assignment 2.* We aim to provide a language that is at least as expressive as required to translate to a high-performance implementation of the circle renderer from assignment 2 with minimal code written. On the user end, this includes processing and organizing data over several different representations and merging together multiple streams of data.

- *Write a compiler.* We aim to provide a sound translation from all representable programs in our language to CUDA, which is then compiled with NVCC. This will involve generating both host and device code in order to provide a complete program, and managing the movement of data between the two. This will to some extent enable the generation of heterogeneous code.

- *Develop heuristics.* We will develop heuristics to analyze program properties and determine if we can apply CUDA optimizations such as shared memory, kernel launch minimization, constant and texture memory, and optimize using general performance tuning metrics.

- *Write example programs.* We will demonstrate several reasonably complex programs expressed and compiled into our language and analyze the resulting performance from the compiled (parallel) code.

- *Demo.* A functional demo of our project should allow users to quickly mock up a parallel program inside the GUI, visualize its structure, compile it, and gain a reasonable understanding of its parallel performance within the interface without knowledge of, or interaction with, CUDA (or any other parallel back-end.)

6.2 Stretch Goals

- *Write more advanced primitives.* If all goes well, we plan to expand the set of provided primitives to include things like scans, sorts, and other basic operations such as those available in the Thrust library.
- *Analyze caches.* We may also choose to perform additional experiments in analyzing cache traversal patterns (developing a cost semantics for the language), and subsequently simulate and compile equivalent programs with more optimized cache behavior.
- *Target more backends.* We may choose to target other parallel primitive back-ends (such as message-passing) in order to distribute our language's workload optimally over a machine or cluster.
- *Achieve performance comparable to a CUDA programmer.* In an optimal case, our language will be able to compile to code that is roughly as high-performance as that written by a human programmer with some knowledge of CUDA, such as the threshold expected by a 15-418 assignment's requirements.

7 Platform

CUDA is a platform that both partners are familiar with, and its computation unit (kernel launches) will be interesting to analyze in the context of a programming language with explicit data dependencies.

8 Schedule

- Week of 10/29
  - Write proposal
  - Design semantics of core language
  - Write parser for core language representation
  - Background reading on compilation strategies for data-parallel languages.
- Week of 11/05
  - Write various versions of circle-renderer code at various levels of abstractions using core language semantics.
  - Basic CUDA backend for language subset.
  - Finalize list and implementation of language primitives.
- Week of 11/12
  - CUDA backend for core language
  - GUI frontend and compilation to core IR
  - Write checkpoint report.
- Week of 11/19
  - Develop kernel-launch-minimization optimisations.

  - Develop and test benchmark programs
  - Flesh out GUI and user-defined function integration.

- Week of 11/26
  - Develop shared-memory optimisations and analyses.
  - Analyze performance of translated CUDA with respect to benchmark programs.

- Week of 12/03
  - Refine previous optimisations.
  - Develop cache-performance optimisations.
  - Begin demo program.

- Week of 12/10
  - Write final report.
  - Polish demo programs.