Final Report
15-418 Fall 2018

Nick Roberts, Dan Cascaval
nroberts, dcascava

1 Project Information

Title Dag: A data-parallel, domain-specific programming language
Website `https://nick-and-dan.github.io/418/`
Repository `https://github.com/ncik-roberts/dag`

2 Summary

We designed and implemented a high-level, domain-specific data-parallel programming language based off of the notion of modeling program structure as a directed acyclic graph. This language benefits from a variety of parallel primitives available to the user that provide alternative compilation strategies (among which the compiler can choose when constructing the output program). To demonstrate the potential of this language for parallel computation, we compile a variety of benchmark programs to CUDA. We emphasize the ease of development in this language, and claim that a programmer can implement correct programs that take advantage of the majority of available parallelism in a target with less effort.

3 Background

3.1 The state of GPU programming

GPUs provide programmers with an opportunity for a great deal of parallelism, and CUDA is a popular programming language for harnessing this parallelism. The fundamental building block of a CUDA program is the kernel launch, where the programmer indicates a task to be run massively in parallel. A kernel launch is an information boundary: the programmer explicitly copies whatever data on the host must be made available on the device; and, following the conclusion of the launch, the programmer retrieves data from the device by another explicit copy. The programmer designates what portion of data to process within the task itself, with the executing core recovering its designated portion by doing arithmetic on the thread index and block index.

A programmer is therefore faced with several decisions when writing a CUDA program:

- A program normally consists of nested iteration patterns over data. Which of these iterations should be converted to kernel launches, and which of these should be run sequentially on the host?
- Later computations in the program use the result of previous computations. Should additional buffers be allocated to store the result of earlier computation, or can the later computation recover the results without needing to read from memory?

Once a programmer has fixed an iteration scheme and decides what memory needs to be allocated, she must provide a correct implementation of several things:

- Determining the partition of data associated with each instance.
- Determining which data must be sent between the host and the device.

When tuning performance, for each configuration of decisions, the programmer has to reimplement this decomposition and data transfer. As such, trying out multiple configurations of iteration and allocation schemes is costly. In addition, in our experience, the indexing code is a source of hard-to-find arithmetic errors when, in theory, the designation of most static partitions should be automatically derivable from the indexing scheme.

Our project is concerned with reducing the cost of considering the many alternatives of parallelizing a program with CUDA. Given a specification by the programmer, we do this by performing a search over the possible *decisions* listed above, automatically generating the *implementation* associated with that decision. The specification by the programmer is in the form of a program, naturally, in a high-level data-parallel language; the output is a CUDA program that maximizes some heuristic.

3.2 Example program

We name our language **Dag** for reasons that, if not self-explanatory, will be made clear shortly. To motivate further discussion, we include the source text for an example program for performing matrix-matrix multiplication:

```
int[][] multiplyMatrixMatrix(int[][] m1, int[][] m2){
  return for (int[] row : m1) {
    return for (int[] col : transpose(m2)) {
      return reduce(+, 0, zip_with(*, row, col));
    };
  };
}
```

The structure of a program is comparable to C, with two major departures: (1) the exclusion of `for`-loops in favor of `for`-*expressions* that denote performing a series of computations at each element of an array, and (2) the inclusion of parallel primitives, like `reduce` and `zip_with`.

The `for`-expression is the building block of a dag program. It is analogous to a "map" operation whose body can refer to variables in the enclosing scope. This is a limited form of closure. The result of evaluating an expression `for (type x : expr) { stmts }` is the array consisting of the return value of the `stmts` body on each element `x` of the array `expr`. In the above matrix multiplication example, the outer `for` loop iterates over `m1`, binding the variable `row` to stand for a row of `m1`. The outer loop immediately returns another `for`-expression; this one binds a variable `col` to stand for each row of the transposition of `m2` (i.e. each column of `m2`), and immediately returns the result of summing the products of corresponding elements of `row` and `col`. Hence, the return value of the outermost `multiplyMatrixMatrix` function is a 2D array each of whose elements is the result of running `reduce(+, 0, zip_with(*, row, col))` for the appropriate substitution for `row` and `col`.

3.3 Structure, and data structures, of output CUDA program

When discussing the data structures of our project, it's our duty to talk both about the data structures we used in the compiler and the *meta* data structures used in the output CUDA program's runtime. The CUDA data structures bear more relevance to parallelism, but for completeness, we discuss both.

The core data structure is *buffers* to hold the result of the massively-parallel operations indicated by `for`-expressions and parallel primitives in the source language. These buffers, unlike the apparently-multidimensional arrays in the source language, are single-dimensional to

avoid indirection. The `dagc` compiler automatically inserts instructions to allocate and transfer these buffers based on the structure of the source program.

We allow for user-defined data structures, such as structs, to increase the expressivity of the language, but this is for convenience, not out of necessity.

Paradoxically, the data structure we use to greatest success is the *absence* of a data structure. Many of the arrays that appear in the source program end up not being represented at all in the generated CUDA. Consider again the matrix multiplication example: it would be wasteful if the call to `transpose(m2)` were performed in memory at all, let alone for each iteration of the outer loop. Enter the *array view*. An array view is an array that, despite being explicitly represented in the input program, will be turned into an iteration scheme in the output CUDA. That is, the data apparently stored in an explicit Dag array is instead computed as needed in a CUDA loop.

Absent parallelism, and absent explicit arrays, the matrix multiplication example will compile to a C program that roughly looks like:

```
/* M1, N1, M2, and N2 would also be passed as the height and
 * width of m1 and m2, respectively */
void multiplyMatrixMatrix(int[] output, int[] m1, int[] m2) {
  for (int i = 0; i < M1; i++) {
    for (int j = 0; j < N2; j++) {
      int acc = 0;
      for (int k = 0; k < M2; k++) {
        acc += m1[i * M1 + k] * m2[k * M2 + j];
      }
      output[i * M1 + j] = acc;
    }
  }
}
```

There are some differences to note between the input program and the output program:

- The `int[][]` return value is instead represented as an output buffer, `output`, which is updated incrementally as results are calculated.
- The `int[][]` parameters are instead flattened to `int[]` to lessen indirection.
- The transposition of `m2` manifests only when the C code indexes into `m2`: as you can see in the access `m2[k * M2 + j]`, the indices `j` and `k` are reordered.
- The `reduce` is expanded into an explicit iteration that accumulates the result into `acc`; the `zip_with` argument to `reduce` manifests only as the body of the for loop, where `*` is applied, in sequence, to the corresponding elements of the arguments to `zip_with`.

Parallel operations and their associated semantics Table 1 describes the parallel operations available to the user, along with the indexing scheme used to access the ith member of the array view if the compiler chooses to not explicitly represent the array in the CUDA program. (The sequential implementation for each of these is the obvious for-loop, so we do not elaborate on this.)

Operation	Semantics	Parallel impl	Indexing scheme
`for`$(\tau\ x : xs)\{stmts;\}$	Map the statements over each element x of xs, with the return value of the statement block being assigned to the corresponding location of the output.	Kernel launch.	N/A
`zip_with`$(\texttt{op}, xs, ys)$	Create an output array by pairing corresponding locations in xs and ys and applying op.	Kernel launch.	$xs[i]$ `op` $ys[i]$
`filter_with`(xs, bs)	Create an output array consisting of the members of xs where the corresponding member of bs is set to `true`.	N/A.	N/A
`reduce`$(\texttt{op}, id, xs)$	Combine all elements of xs with `op`, where id is the identity of the operation.	Thrust call.	N/A
`scan`$(\texttt{op}, id, xs)$	Create an output array consisting of the result of reducing every prefix of xs with id and `op`.	Thrust call.	N/A.
`range`(n)	Create an array with elements from 0 to $n-1$.	Kernel launch.	i
`transpose`(xss)	Create the transposition of xss.	Kernel launch.	$xss[j][i]$

Table 1: Parallel operations and their semantics

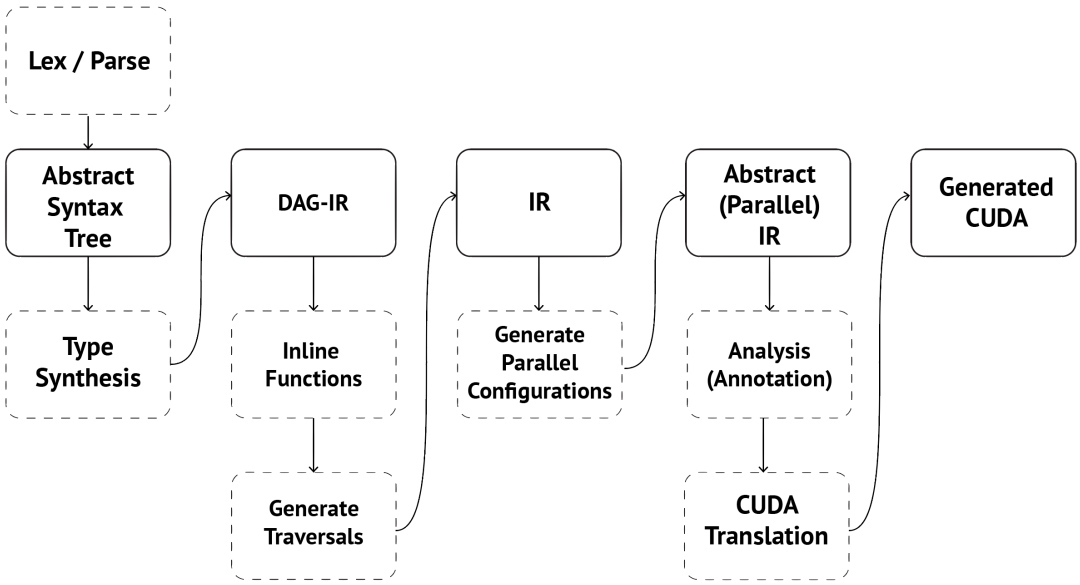
3.4 Structure, and data structures, of compiler

The compiler's main interesting data structure is the eponymous directed acyclic graph that represents a program's control flow. From this dag representation, the compiler generates all possible traversals of the dag, and, following that, generates all possible *parallelizations* of each traversal. More concretely, the compiler follows these steps:

1. Creates a dag representation of the source program.
2. Generates all topological sorts of the dag.
3. For each topological sort, considers many possible alternatives for parallelizing the program, including fusing loops, parallelizing inner loops, and parallelizing outer loops.
4. Choose the best available alternative based on some heuristic.

The astute reader will wonder why it is the Dag compiler's responsibility to perform the topological sort, since it hands off the result to `nvcc -O3` ultimately, anyway. The reason is this: a kernel launch is an information barrier across which `nvcc` will not perform optimizations. The Dag compiler must therefore consider alternatives where computation is performed prior to a kernel launch versus within a kernel launch; one alternative may result in a better CUDA program. Hence, `dagc` should indeed generate a dag.

Flow of compiler The compiler is structured as follows:



1. **Lex, Parse; AST; Type Synthesis.** The standard compiler frontend, this converts the unstructured source string provided by the user to a structured, typed tree that can be analyzed by the rest of the compiler.
2. **DAG IR.** The compiled program is turned into an explicit dag representation. In this representation, the user-written functions are inlined until there are no remaining function calls (which is always possible, absent recursion).

3. **Generate Traversals → IR.** To reach this intermediate representation, all possible traversals (topological sorts) of the dag representation are considered. Each topological sort generates its own IR tree. In this IR phase, we still allow for nested parallel `for` blocks, which is not feasible in the output CUDA (since it's not possible to launch a kernel from within a kernel launch).

4. **Generate Parallel Configurations → Abstract IR**. This transformation removes the nested parallel blocks. There are, in general, multiple ways to remove nested parallel blocks from a program while retaining the same semantics; this phase generates many possible such configurations.

   An optimization we perform is the "fusing" of nested `for` blocks. Consider the loops:

   ```
   return for (int x : xs) {
     return for (int y : ys) {
       ...
     }
   }
   ```

   In CUDA, we can imagine wanting to assign a task for each possible pair (x, y) rather than only parallelizing along xs or along ys. To accomplish this, we permit in the Abstract IR phase for this program to take on form that looks like this:

   ```
   return for (int x : xs, int y : ys) {
     ...
   }
   ```

   which returns an $|xs| \times |ys|$ two-dimensional array where the body of the `for`-expression is performed for each pairing (x, y).

5. **Analysis (Annotation).** In order to perform the final translation to CUDA, we must collect a great deal of data about the Abstract IR form. In particular: the *expressions for indexing into array views* (which allow the virtual iteration over array views without explicitly allocating the array), the *lengths of all arrays in the program* (which allows for flattening multidimensional array indexing into single-dimensional array indexing, as well as the appropriate sizes for `memcpy` and `cudaMalloc`), and *free variables in kernel launches* (which are those variables which must be copied from host to device when launching the kernel).

6. **CUDA Translation → CUDA.** Generate CUDA for each Abstract IR tree generated from the previous phase. Apply a heuristic, like total amount of memory allocated and arithmetic intensity, to select the best CUDA implementation.

3.5 What stands to be gained

A question that arises in many parallel applications—what parts of the program stand to benefit from parallelization?—takes on an interesting form when applied to a compiler. On one hand, the answer is almost passé—the amount of parallelism the user will see in the output CUDA is entirely dependent on the amount of parallelism made available in the source program via use of parallel primitives and parallel `for` loops. On the other hand, it is interesting to consider what *class* of input programs see good speedup when run through `dagc`. Based on our benchmarks, reported in the Results section, the answer appears to be those programs that perform many nested parallel `for` loops over data where the inner loop performs some intensive arithmetic computation. This is consistent with the intuition that applications with high arithmetic intensity should see good speedup.

3.6 Workload in the generated CUDA

Just as the language is data-parallel, so too is the generated CUDA. The kernels frequently consist of a loop where a regular arithmetic computation is performed to the task's designated element of a large input array. In fact, the highly regular style of parallelism encouraged in GPU programming made a data-parallel language very attractive as the source.

4 Approach

4.1 Technologies used

The compiler is written in OCaml, version 4.05.0. If you wish to build and run the compiler yourself, installation instructions for all necessary packages are included in the README of the GitHub repository.

The compiler generates CUDA that can be compiled with `nvcc` version 2.0 or later. We additionally make use of the Thrust library for some parallel primitives, like scan. Benchmarks for the output of the compiler were run on an AWS `g2.2xlarge` instance. This runs CUDA on one GPU in a Grid K250 GRID board, with 1536 cores, 800MHz core clock, 4GB of GDDR5 RAM, and a PCI-e x16 Gen3 interconnect.

A testing harness is included with the GitHub repository that makes it easy to verify the correctness and performance of the generated CUDA.

4.2 Mapping the user program onto parallel hardware

The early phases of the compiler described in Section 3.4 work well to identify available parallelism; the translation into CUDA is concerned with mapping that available parallelism onto GPUs. If an expression of the form `for (type x : xs) { stmts }` is translated into a kernel launch, elements of `xs` are blocked into contiguous sections of memory and assigned, in sequence, to CUDA threads.

The contiguous blocking of the input data, in addition to achieving high locality within a single thread, is ideal for CUDA's mapping of threads onto warps. Our kernels generally do not contain branches (only doing so in the presence of `filter_with`), and so the shared instruction stream among the 32 threads on a single warp can easily be executed in lockstep.

4.3 Iteration process

Writing an end-to-end compiler is no easy task, so our iteration effort was concentrated on devising performant solutions to the issues that arise in writing a translation. Below, we list some of our iterations, including the performance problems we encountered that necessitated the iteration.

1. **Improve coverage of traversals.** An important aspect of our compiler is the fact that it considers many possible traversals of the program's dag before deciding on the most efficient one. The topological sort of the input program is non-trivial because a `for`-block is a vertex of the control flow graph that has a nested control flow graph of its own, in the form of the statement body which will be translated to a kernel launch. Earlier iterations of our compiler never traversed the nested control flow graph before evaluating the `for`-expression itself; we improved later iterations to allow for the extraction of instructions from inside a for block.

   For example, take the program:

```
return for (int x : xs) {
  int y = reduce(*, 1, ys);
  return (int z : zs) {
    ...
  }
}
```

Later iterations of our compiler will try a traversal where the statement `reduce(*, 1, ys)` is performed prior to the kernel launch over `xs` since the statement does not depend on `x`.

2. **Prioritize highly-nested parallelism.** In this iteration, we decided which heuristics to prioritize when selecting the "best" output CUDA program out of the alternatives. We found that the loop fusion described in section 3.4 of the loop was highly effective in improving speedup. Multiplying two 1024×1024 matrices decreased to 0.911s from 1.34s when prioritizing the highly-nested parallelism.

3. **Use Thrust primitives for scan and reduce.** Thrust's inclusive scan operation provides pre-packaged parallelism with exactly the desired semantics of our language's scan and reduce, so allow the compiler to generate candidate programs where these operations are performed in parallel. Earlier iterations would just perform these sequentially.

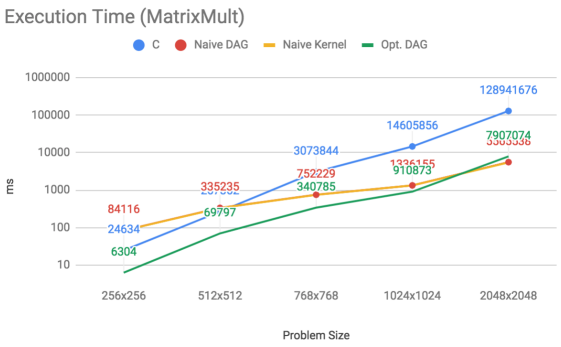
4. **Keep track of sizes of filtered data.** In the presence of nested for loops that return a filtered array, it is necessary to allocate extra buffers to store the filtered length of the returned arrays. Implementing this iteration did not improve performance, but it did improve the flexibility of where filtered data could be created and consumed.

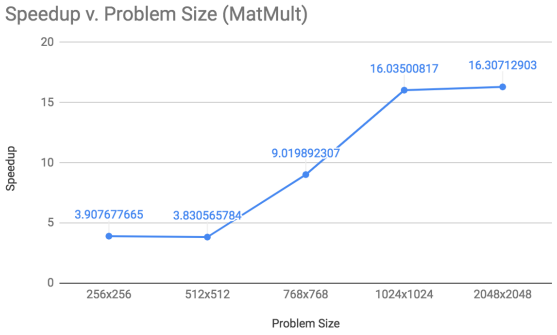
5. **Block data.** Assigning contiguous blocks of elements to each CUDA thread rather than single elements improved our benchmark programs' efficiency, especially the calculation of the Julia set.

5 Results

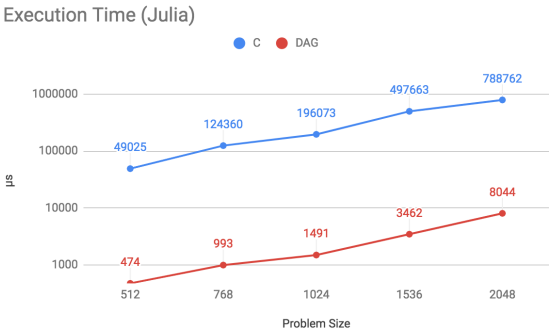
We benchmark the compiler's performance by measuring the execution performance of the program that the compiler outputs for a given source program. To analyze the compiler's heuristics, we show benchmarks using a "naive" output (Bad), versus an optimized output (Good). We have structured our compiler such that it intentionally considers and optimizes programs that are likely to display better parallelism first. This is more apparent in the complex examples with a wider decision space. We benchmark against a vectorized C implementation on a program with a lot of communication, to display our blocking strategy (Matrix Multiply), a program with potentially low arithmetic intensity (Convolve), and a program with high arithmetic intensity (Julia). Additionally, we sometimes isolate benchmarking to the kernel launch itself, trying to get a sense of the performance irrespective of memory transfer overheads.

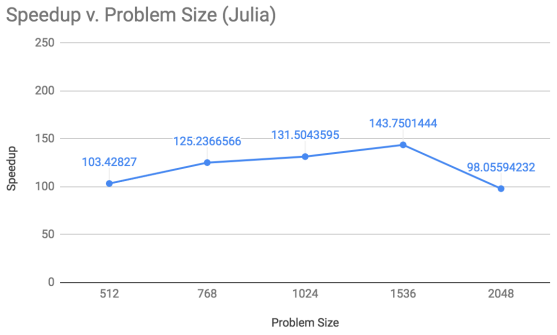
On MatrixMultiply, we notice that our test program does not benefit from our naive CUDA implementation at small input sizes. Initially we theorized that this results from overhead of GPU access and communication, but upon benching the program we found that the kernel in fact takes up almost the entirety of this time. Upon implementing heuristics in our compiler (merging nested parallel blocks,and implementing blocking), we notice much better output, gaining a performance increase over the C version that increases significantly as the input size passes 2MB and the naive version starts missing the L2 cache.

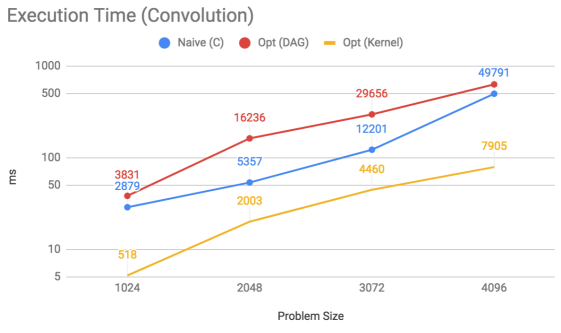


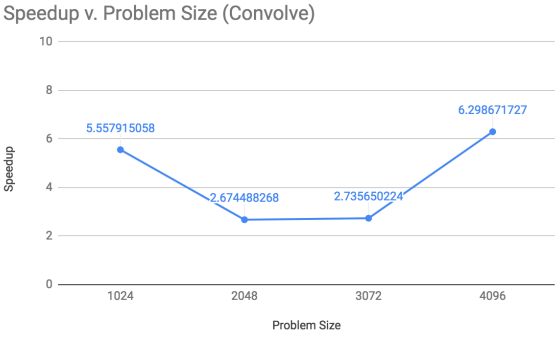


In the Julia example, we benchmark a program to render a monochrome image of the Julia set (shown at the foot of this report). This shows our project's performance at its best—rendering the Julia set is an embarassingly parallel problem with high arithmetic intensity (lots of compute, very little communication), and no branch divergence. It demonstrates our language's ability to use both indexing schemes and merge nested parallel loops. The overheads are minimal—we are transferring very little memory to the GPU—and the numbers demonstrate this large degree of parallelism, with a fairly stable speedup that may eventually fall victim to adverse memory effects at very high resolutions.









In the convolution example, we perform a 3x3 convolution. This has relatively low arithmetic intensity (we perform the 9 arithmetic operations sequentially for each pixel), and we see a lot of overhead occurring here, as the kernel outperforms the naive implementation significantly, but the total overhead including transfer time and memory allocation is a much larger percentage of the execution time than in MatrixMultiply or Julia. Additionally, we do not perform well with CUDA's abstractions: since we store data in 1D arrays to lessen indirection, our method of mapping the convolve creates a lot of communication and accesses with poor spatial locality.

Our performance improves (relative to the naive version) at larger sizes as the same factors take effect in that version as well.

5.1 Analysis

There are a number of factors to consider when optimizing GPU performance integrating into source code. As we compile and link to entirely host code (the programmer never works directly with the parallel hardware), we incur a significant overhead in transferring and allocating memory for smaller programs with low arithmetic intensity. We attempt to minimize memory allocation and copying wherever possible within the generated DAG code itself by using the array views and indexing schemes discussed above, but these factors can weigh heavily in smaller benchmarks. We also consider compile-time performance as a limiting factor—when the search space of possible traversals and generated programs is very high, we cannot feasibly evaluate all of the options, and our compiler will only consider a limited subset of possible programs that are likely to be better in order to terminate in a reasonable amount of time.

Additionally, we note that the examples considered here all represent fairly regular parallelism, as the language constrains the expression of irregularly parallel workloads.

6 Work Distribution

Nick Roberts and Dan Cascaval were the contributors to this project.

Nick took the lead on many aspects related to the design of the source language and the construction of a translation to efficient CUDA. To be more concrete, he:

- designed the structure and semantics of the source language in a way that made parallel abstractions available at low cost to the programmer,
- created efficient translations of array views to avoid unnecessary memory allocation,
- implemented language features such as the parallel `for` expression (a kernel launch), filtering (keeping track of lengths of filtered arrays in memory for future use), reduction, and scan,
- created the explicit dag representation of the program and wrote a topological sort that, in particular, allowed for the extraction of instructions out of kernel launches, and
- designed data structures for the compiler phases, most prominently the CUDA translation phase where it is necessary to keep track (statically) of lengths and virtual indexing schemes for all array views used in the program.

In addition to working on various aspects of the compiler, Dan took the role of benchmarker and performance engineer. He:

- incorporated parallel implementations of scan, reduce, and transpose into the translation,
- implemented miscellaneous language features, like structs, arithmetic, and range expressions,
- wrote several programs in the source Dag language, such as the Julia set, circle renderer, convolutions, and various stress/correctness tests for language features.

- wrote a testing harness for determining the performance and correctness of a variety of Dag programs against sequential and vectorized C versions, and
- ran benchmarks and performed manual editing against the CUDA translation to inform which heuristics to use for choosing among alternatives and to identify sources of slowness.

Each team member's contributions complemented the other's. The contributions were of a different nature, and it is imprecise to try to compare across natures, so instead we compare effort. We roughly designate 50-50 credit between the partners.

Julia Set, rendered by one of our compiler's benchmarks.