CS533 Project Progress: Map Reduce

Abdul Dakkak, Carl Pearson, Liwen Chang

Project Summary

Our project proposal stated that we would be implementing a compiler for a functional language that would generate MPI code that would run on a cluster of machines and across threads in a shared memory system. We have narrowed down our objective to generating code that will run on the GPU. The interesting features in the compiler and language, as stated in the proposal, is that parallelization will be done via Map and Reduce operations. The compiler will maintain the IR for these operators and will perform optimizations on them to generate performant code.

It is worth mentioning that our aim is not to write an optimized compiler (the compiler can be very slow), our aim is to have a compiler that can generate optimized code. To that end, we have written the compiler in Dart (a Javascript inspired language) that currently generates sequential Javascript code from our language. The Javascript generation is primarily meant for debugging purposes and the next steps would be to generate threaded CUDA code and perform some compiler optimizations on the IR.

The next few paragraphs revise and summarize the project as well as detail what has been implemented and what will be implemented in the future. At the end we give a schedule for the project.

Language Details

Our statically typed language is inspired by array and data flow programming languages such as Fortran[1], APL[2], and LINQ[3] where one expresses computation based on operations on vectors and arrays. In our language, one can define two vectors of size 100 by

```
n :: Integer = 1000;
as :: []Real = rand.Real(n);
bs :: []Real = rand.Real(n);
```

We can then add the two vectors using an overloaded plus operator

```
res :: []Real = as + bs;
```

To give you a taste of the language, in this program we approximate `pi` using Monte Carlo integration

```
def f(a :: Real, b :: Real) :: Bool {
return a*a + b*b < 1;
}
res :: Integer = zip(as, bs).count(f) / n;
```

this would be translated into the map/reduce operations of

```
t1 = map(f, zip(as, bs));
count = 0;
reduce((x) => count += x, t1);
res = count / n;
```

The compiler would lower the above into an IR representation, maintaining the map/reduce operations

```
Instruction(zab, zip(as, bs))
Instruction(t1, Map(f, zab))
Instruction(count, 0)
Function(g, (x) => count += x)
Instruction(r2, Reduce(g, t1))
Instruction(res, divide(count, n))
```

The compiler is able to analyze the above code, inline the map operation into the reduce function and privatize the `count` variable. This results in efficient code that can be parallelized and does not produce unnecessary temporary arrays.

We plan on expressing all commonly used analytics operations such as sort, mean, max, min, histogram, variance, etc... in this framework.

Compiler Pass

In this section we discuss some of the compiler transformations we will be examining. These optimizations have not been implemented.

Loop Fusion

The most important factor in parallel computing is how to manage memory transfer. If a node computes a chunk of data and it is used in subsequent instructions, then it should reuse the output rather than send and request the data again. There are two approaches to facilitate this. The first is a runtime approach: Hadoop, for example, dispatches tasks to maximize reuse of local data. This done via the Hadoop scheduler which has a mapping between nodes and data state.

The second is a compiler transformation. This is mainly done via loop fusion. If for example, one writes a program `map(f, map(g, lst))` then a compiler pass can transform this into `map(f.g, lst)` where `f.g` is the composition of `f` and `g`. A simple peephole optimizer can scan for this instruction pattern and perform this transformation. A generalization of this technique for other list primitives is found in the the Haskell vector library. Using a concept called Stream Fusion[4], Haskell fuses most function loops to remove unnecessary temporaries and list traversals. In this project, we will adopt some aspects of how Haskell performs this transformation when they are applicable in the CUDA programming model — since GPU programs tend to be memory bound, reducing the number of temporaries increases performance, for example.

In order to achieve high-performance fusion, a resource estimator and a memory counter are necessary. The resource estimator monitor on-chip resource usage for a current tiling to avoid register spilling and drop of multithreading. The memory counter simple calculate the potential memory count saved for fusion. Since 1) on-chip resource of GPU is limited, and 2) the performance gap between on-chip access and off-chip access on GPU might has less impact than the gap between on-node access and inter-node access on cluster, the decision of fusion may not be trivial. Here, we will heuristic

function of these two parameter to determine the decision of fusion. A precise resource estimator might be very difficult to implement and may not really meet the real vendor compiler. In this project, we will simply build a routine to rely on the feedback of the real vendor compiler after the GPU backend generated the code.

Loop Tiling

Tiling is critical for performant on GPU programs. In this project, automatic tiling will be considered as group several single function `f`. Considering function `f` is isolated and map limit data dependency among different output, an analysis can be done for function `f` to track data sharing among multiple outputs. Here, sharing analysis is considered as access pattern analysis of function `f`. Therefore, tiling for data sharing is achievable.

Coalesced access of GPU is also important for performance. In this project, coalesced access is easily considered as a specialized type tiling. Register packing and thread coarsening of GPU are also recognized by applying the same analysis, with different tiling. Therefore, we can potentially perform very aggressive tiling for GPU.

Autotuning

One of the advantages of compiling from a high level languges into CUDA is that you can easily tweak parameters for loop tiling, unrolling, and fusion. A combination of a resource model and autotuning will be used to maximize the performance of the generated code. We will use a bruteforce like algorithm, similar to the one employed by ATLAS to explore the parameter space.

Implementation Details

To facilitate rapid prototyping in this project, we chose the Dart programming language. The Dart language developed by Google is a modern interpretation of Javascript — a cross between C++/Java and Javascript. It adds classes, types, and polymorphic instances (via a templating mechanism) and is able to compile down to Javascript or can be run in the Dart VM. The language also has good documentation, extensive standard libraries, and an active library development community (the parser generator, for example, is a library we are using).

Due to the structure of our compiler, it is backend-agnostic. Currently, we generate sequential Javascript for debugging purposes, but have a prototype CUDA backend that generates naive code. In the next few weeks, we plan on refining our CUDA implementation to hide memory copy latency and optimize for the correct launch parameters for the kernel.

Hardware

Our generated code will be tested against NVIDIA Fermi C2050 and Kepler K10 GPU architectures both of which we have access to.

Evaluation

We will use parboil as our benchmark suite, picking 4-5 benchmarks that map nicely to our language. We will then measure the performance obtained from our compiler versus hand optimized GPU code found in the Parboil benchmark suite. We will also compare the ease of programming in our language versus native CUDA.

Progress Summary

We have developed the infrastructure to allow us to start working on the interesting parts of the project. During the rest of the semester, we will develop compiler passes that allow us to generate efficient backend code. Currently, we have the following in our compiler/language implementation:

- Language Parser
- Instruction lowerer
- Naive backend to generate CUDA code
- Backend to generate Javascript code (this is used for debugging)
- A framework to allow us to perform analysis (a visitor for instruction sequences and blocks, for example)

The following table is our projected timeline for the rest of the semester.

Week	Task
3/17	Finish naive CUDA code generator.
3/24	Add compiler pass to perform closure conversion (for lambda functions) and calculate the `def-use` chain of variables.
3/31	Generate optimized map kernels (this requires finding tuning parameters for architectures and NVIDIA provides tools to programatically determine those parameters).
4/07	Generate optimized reduce kernels (this, again, requires some tuning, but a group member has done extensive research on reduce operations on GPUs).

4/14	Add compiler pass to perform function fusion.
4/21	Experiment with other compiler passes, such as loop unrolling, that would increase the compute work done by each thread.
4/28	Final benchmarking and project writeup.
5/05	Complete project presentation.

Projected timeline for the project along with the associated tasks.

References

[1] J.C. Adams, W.S. Brainerd, and C.H. Coldberg, *Programmer's guide to fortran 90*, Intertext Publications, 1990.

[2] R.P. Polivka and S. Pakin, *APL: the language and its usage*, Prentice Hall Professional Technical Reference, 1975.

[3] E. Meijer, B. Beckman, and G. Bierman, "Linq: reconciling object, relations and xml in the. net framework," *Proceedings of the 2006 aCM sIGMOD international conference on management of data*, ACM, 2006, pp. 706–706.

[4] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion: from lists to streams to nothing at all," *ACM sIGPLAN notices*, ACM, 2007, pp. 315–326.